The FM9001 Microprocessor Proof

Bishop C. Brock, Warren A. Hunt, Jr.,
Matt Kaufmann

Technical Report 86 December 23, 1994

Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas 78703-4776

TEL: +1 512 322 9951
FAX: +1 512 322 0656

EMAIL: hunt@cli.com

This work was supported in part at Computational Logic, Inc. and by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc.

Contents

List of Figures

ABSTRACT

The FM9001 is a 32-bit, general-purpose microprocessor whose specification and netlist-level implementation have been formally described in the Nqthm logic. The FM9001 netlist has been mechanically checked to implement its specification with the Nqthm theorem-proving program. This report includes the entire formalized input: specification, implementation, and theorems, all of which have been checked with Nqthm. The FM9001 microprocessor has been fabricated as a CMOS gate array by LSI Logic, Inc. We have been testing the fabricated FM9001 for over two years and we have not yet discovered any flaws. Also included in this report is the netlist from which LSI Logic fabricated the FM9001. This report is intended to permit an Nqthm user to completely redo the FM9001 proof, and also to formalize and verify other hardware in the `DUAL-EVAL` hardware description language.

Keywords: hardware verification, formal specification, hardware description language, circuit synthesis, CLI short stack, formal methods, FM8501, FM8502, FM9001, Piton

Chapter 1

OVERVIEW

The FM9001 microprocessor is a general-purpose 32-bit device whose gate-level netlist design implementation was developed using a theorem-proving environment in conjunction with a traditional CAD system. This netlist was translated to LSI Logic's NDL (Netlist Description Language) for physical implementation. The behavioral specifications for the FM9001, the definition of the hardware description language (HDL) used to represent the design of the FM9001, the simulator for the HDL, and the verification of the FM9001 were all carried out using the Boyer-Moore theorem-proving system Nqthm.

This document presents the details of the FM9001 development, specification, and verification. First, we give a little history of the FM9001 project followed by a summary of our results. The remainder of this document is devoted to giving the details of the specification and verification of the FM9001 from an Nqthm user's point of view. Thus, we imagine that the reader probably has some familiarity with Nqthm.

The FM9001 behavioral and design specifications, the hardware description language and its semantics, the FM9001 correctness proof, and the preparation of the FM9001 for fabrication were performed entirely by Bishop C. Brock and Warren A. Hunt, Jr. The FM9001 microprocessor was manufactured by LSI Logic, Inc., who also performed post-manufacture testing [1]. More thorough testing was performed by Kenneth Albin by building a purpose-built single board FM9001 computer with an interface to a logic analyzer. Lawrence Smith interfaced the FM9001 to Tektronix chip tester, which allowed much more complete testing. Matt Kaufmann proved the ALU interpretation lemmas (Section 13.2), and assisted with proving reset properties (Sections 2.13, 11.3, 12.1). Ann Siebert did much of the work on the recognizer for well-formed circuit descriptions (Section 5.4). The arithmetic libraries were taken largely from existing libraries at Computational Logic, Inc. (see Chapter 3).

The first draft of this report was written by Warren A. Hunt, Jr. and Matt Kaufmann. The report was subsequently edited by Warren A. Hunt, Jr. and Robert

S. Boyer. Ann Siebert wrote Section 5.4, and William D. Young helped with Section 5.1.

All of this work was performed at Computational Logic, Inc., as a research project. This effort was supported in part by ARPA.

1.1 HISTORY OF THE PROJECT

The FM9001 effort started in earnest when we (Bishop Brock and Warren Hunt in November of 1989) defined a simulator, which we called `DUAL-EVAL`, for a language that was general enough for representing the design of a microprocessor along with its memory system [10, 15]. Representing circuits as data was a turning point in our efforts at hardware specification. Previously Hunt [13, 14] had represented hardware with functions in the Boyer-Moore logic. We also investigated representing circuits as predicates [8], in the tradition of the HOL hardware verification community [11]. Neither of these approaches permitted operating on a "circuit" directly; that is, instead of having a constant that could be interpreted we were trying to represent circuits as expressions in a logic. Why it took so long for us to realize that neither approach would be satisfactory (as evidenced by the CAD community's approach to circuit representation) is difficult even for us to understand.

The point of representing circuits as data as opposed to representing circuits with functions can be likened to representing circuits as a netlist in a purpose-built netlist language as opposed to representing circuits as C-language code. That is, we want to use the C-language code to implement simulators, recognizers, synthesizers, etc., for operating on our netlists, and not attempt to implement a netlist as a C-language program—the CAD community has operated with this paradigm for many years. Thus, when we say we have defined a hardware description language (HDL) we mean that we have defined a recognizer function (in the Boyer-Moore logic) that determines whether some netlist is well formed and a simulator function (also written in the Boyer-Moore logic) that, given input data, simulates well-formed netlists. Representing circuits as data also permits the verification of tools used to operate on the language; that is, it is possible to prove the correctness of circuit synthesis functions. However, mechanically proving the correctness of large circuit designs by reasoning about their simulations is more complicated and difficult than our previous approach that used direct representations in the logic.

Immediately prior to the definition of the `DUAL-EVAL` simulator and language recognizer we had defined several different hardware description languages which differed in representational power. For instance, one such language (with associated formal simulator) was called `WAVE-EVAL`, which could simulate a less restrictive class of circuits than `DUAL-EVAL`. However, it seemed too hard to reason about. Another language/simulator that we defined was named `HEVAL`[9], but this early language was restricted to combinational logic circuits only. Once we had defined `DUAL-EVAL`, we spent most of our time building up automated reasoning tools to ease the analysis and verification of circuits described in the `DUAL-EVAL` language, at least until we had automated the process sufficiently to verify the FM9001 design. In actuality, the evolving FM9001 design was a "technology driver" for the development of the `DUAL-EVAL` language. Another part of the HDL development was the evolution of its language recognizer. At first, the recognizer checked for proper syntax and absence of combinational circuit loops; however, over time the recognizer was extended to check for loading and fanout violations, timing properties, well-formedness of LSI Logic names, and a host of other properties. Every time we formalized and verified a circuit, we mechanically translated it to LSI Logic's NDL (Network Description Language), and then with LSI Logic's toolset we compiled the resulting netlist. Each time we found a problem during the compilation or subsequent library linking, etc., we further restricted the class of circuits we admitted by changing the circuit recognizer to disallow the problem encountered. Later Ann Siebert produced a much nicer circuit recognizer that produced error messages; the original recognizer only returned true or false. Another facet of the design process was ensuring that circuits could be tested. Bishop Brock implemented a parallel stuck-at fault simulator that we often used before we even verified a circuit.

The FM9001 was passed off to LSI Logic, Inc., for manufacture at the end of July of 1991. We received FM9001 microprocessor chips some six weeks later. Warren Hunt built a simple test jig that allowed us to check the timing of the various output signals. In 1992, Ken Albin built a single-board computer using the FM9001 microprocessor; this board was constructed in such a way that all of the FM9001 microprocessor signals could attached to a logic analyzer. We ran several programs to check to see if the manufactured part worked as we expected it to [1]. In 1994, Larry Smith interfaced the FM9001 to a Tektronix LV500 chip tester and performed more thorough testing of the fabricated device than was possible with the single-board computer. To this point, we have not found a single operational bug.

Entirely separate from the FM9001 project was J Moore's use of the `DUAL-EVAL` simulator in the succesful specification and verification of a circuit for Byzantine agreement [20]. By using the `DUAL-EVAL` circuit recognizer predicates, he was able to translate his verified circuit into LSI Logic's NDL without needing to understand LSI Logic's tool suite.

1.2 THE FM9001 SPECIFICATION

The FM9001 is specified at four levels of abstraction—user, two-valued, four-valued, and netlist; these are shown in Figure 1.1.

- A high-level behavioral (user-level) model that operates as an instruction interpreter for FM9001 programs;

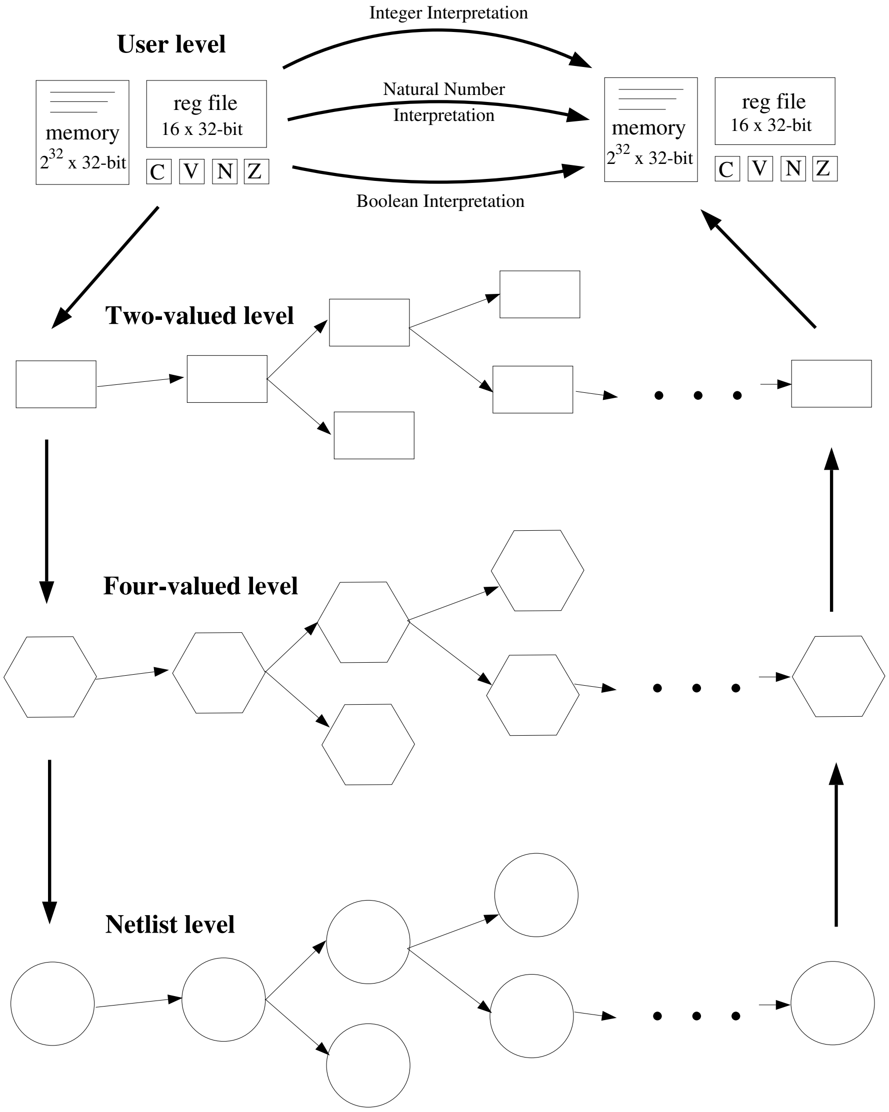


Figure 1.1: Specification Levels

- A Boolean (two-valued) model that "mirrors" the intermediate, register-transfer model;
- An intermediate, (four-valued) register-transfer model that links the low- and high-level specifications for verification purposes; and
- A complete gate-level (netlist) model presented in the `DUAL-EVAL` HDL.

The high-level and the register-transfer models are captured as executable Nqthm logic functions. The semantics for the gate-level implementation, which is described by an Nqthm constant, is given by our hardware description language simulator, `DUAL-EVAL`. Even though we often prove that parts of the register-transfer level are equivalent to the "Boolean level", we do not have a complete Boolean level; this is an artifact of our verification approach. The gate-level model includes all of the logic required to physically construct the FM9001, including the test logic. To have LSI Logic manufacture the FM9001, we translated only the `DUAL-EVAL` netlist into LSI Logic's Network Description Language (NDL) and provided the test vectors required by LSI Logic, Inc.

The implementation of the FM9001 is described as a `DUAL-EVAL` netlist containing a reference for every primitive gate, latch, register, I/O buffer, and wire that is required for the FM9001 implementation. Both the syntax and the semantics of this netlist have been mechanically checked: the netlist is well-formed and it has been proven to implement the FM9001 specification. We have a formal definition of acceptable netlists; that is, we have a predicate that identifies well-formed netlists.

The proof of correctness of the FM9001 gate-level design consists of three major lemmas. First, we have shown that the FM9001 can be forced to a known state, i.e., reset, by a suitable sequence of inputs. This proof is carried out using the gate-level models; the concept of resetting the machine does not appear at the behavorial level of the instruction interpreter. Second, we show that given a set of initial conditions, the gate-level model correctly implements the high-level specification. This proof involves both time and data abstractions to link the clock-cycle based semantics of the gate-level FM9001 model with the high-level instruction interpreter. Finally, we show that the state at the end of the reset sequence satisfies the initial conditions for the previous lemma. This sequence of results provides the formal basis for believing that behavior specified by the instruction interpreter is implemented by the gate-level design. Of course, any physical flaws could invalidate everything. We did, during the design of the FM9001, make provisions for thoroughly testing the manufactured devices.

Physically, the FM9001 is a general-purpose CMOS 32-bit microprocessor. The FM9001 features a two-address programming architecture, five addressing modes, multiple processor bus capability, and extensive arithmetic flag support. At 25 MHz the FM9001 only requires 70 mA of power exclusive of I/O drives; the clock rate may be reduced for lower power consumption. All the latches are connected by a scan chain for testing purposes, except for the register file, which has dedicated test logic.

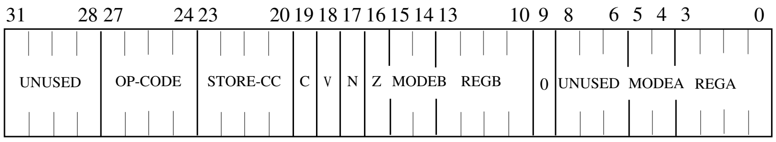
The FM9001 executes instructions by fetching operand A, operand B if necessary, computes a result, and stores the result in the location referenced by operand B, if necessary. Figure 1.2 shows the two FM9001 instruction formats, a two-address instruction or an immediate datum instruction. The two formats only differ in origin of operand A. In the immediate datum case the nine-bit datum in the instruction is sign-extended to 32-bits; otherwise, operand A is selected in the same manner as operand B. Except for the immediate datum mode, operand A and operand B are selected by any of the following modes: register direct, register indirect, register indirect with pre-increment, and register indirect with post-increment addressing mode. Thus, operand A has five addressing modes and operand B has four, where every addressing mode for both operand A and B works with every instruction. The 15 different arithmetic instructions are listed in the table, as are the different conditions that are to be checked for each instruction in order to see if the computed result is to be stored. Each instruction also contains four bits that determine individually whether the arithmetic flags (carry, overflow, zero, and negative) are updated.

The FM9001 programmer's state is shown as a part of the user level in Figure 1.1. This state includes a general-purpose 16-element register file, four arithmetic flags, and the external memory. Register 15 is usually used for the program counter address, but in all other respects operates like the other registers. To perform a conditional branch, operand A is added to the program counter, and the result is conditionally stored based on the `STORE-CC` field of the instruction. Since the results of any instruction may be conditionally stored, there are no differences between control and data operations. This is even true of the program counter because the implementation actually allows any of the 16 registers to be used as the program counter.

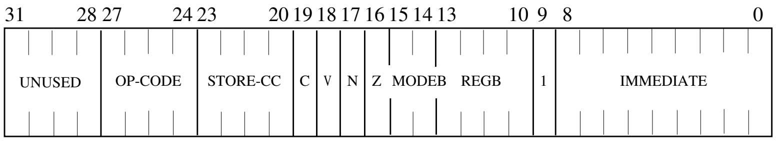
The FM9001 specification was derived from the FM8502 specification [14]. There are three main differences. The FM9001 has the ability to conditionally store every result—only conditional moves were permitted in the FM8502. The FM9001 performs the post-increment operations as each operand is fetched; the FM8502 post-increment operations were done after both operands were fetched and the result was stored. Lastly, the FM9001 has an immediate data mode for operand A; the FM8502 did not have this feature. The ALU operations remained the same as did the fetch-execute cycle. We did this to ensure that the Piton assembler [19] for the FM8502 could be easily transported to the FM9001.

The two-valued and the four-valued specification levels are artifacts of the approach we used to mechanically check the implementation of the FM9001. The two-valued level can be thought of as a Boolean model of the implementation with the tri-state memory interface bus abstracted away. The four-valued level has functionality similar to the two-valued level, except the tri-state memory bus is included; it is at this level that we prove that the FM9001 can be reset. The netlist-level specification describes the actual Boolean gates and latches, test logic, and I/O buffers used to implement the FM9001.

TWO-ADDRESS MODE



IMMEDIATE DATUM MODE



MODE	OPERAND	DESCRIPTION
00	Rn	Register Direct
01	(Rn)	Register Indirect
10	-(Rn)	Register Indirect Pre-decrement
11	(Rn)+	Register Indirect Post-increment

OP-CODE	MNEMONIC	OPERATION	STORE-CC	MNEMONIC	CONDITION
0000	MOVE	b <- a	0000	CC	~C
0001	INC	b <- a + 1	0001	CS	C
0010	ADDC	b <- a + b + c	0010	VC	~V
0011	ADD	b <- b + a	0011	VS	V
0100	NEG	b <- 0 - a	0100	PL	~N
0101	DEC	b <- a - 1	0101	MI	N
0110	SUBB	b <- b - a - c	0110	NE	~Z
0111	SUB	b <- b - a	0111	EQ	Z
1000	ROR	b <- c,a >> 1	1000	HI	~C & ~Z
1001	ASR	b <- a >> 1	1001	LS	C | Z
1010	LSR	b <- a >> 1	1010	GE	(N & V) | (~N & ~V)
1011	XOR	b <- b XOR a	1011	LT	(N & ~V) | (~N & V)
1100	OR	b <- b OR a	1100	GT	(N & V & ~Z) | (~N & ~V & ~Z)
1101	AND	b <- b AND a	1101	LE	Z | (N & ~V) | (~N & V)
1110	NOT	b <- NOT a	1110	T	True
1111	M15	b <- a	1111	F	False

Figure 1.2: FM9001 Instruction Word Formats

Before giving an informal statement of our chief result about the FM9001, we describe that statement in terms of Figure 1.1. The user-level operation of the FM9001 is shown at the top by arrows transforming a memory, register file, and flag state to a new user-level state, which results from executing one user-level instruction. This transformation is shown at the top of Figure 1.1. The correctness of the implementation is stated by mapping a user-level state down to a netlist-level state (see the lower left corner), executing the `DUAL-EVAL` simulator on that state with the FM9001 netlist-level implementation, and then mapping the resultant state back up to the exact same final user-level state. Further, the figure is drawn to indicate that more steps are required to implement an instruction at the lower levels than at the user level.

Here is an informal statement of the chief result proved about the FM9001. Let h be the hardware netlist that we have constructed for the FM9001. We have proved that for each FM9001 user-visible state, s, and for each positive integer, n, there exists a positive integer c such that the result of running the user-model of the FM9001 (the function `FM9001`) n steps can instead be obtained by simulating h and s under the `DUAL-EVAL` semantics for c steps. In precisely stating the theorem, we arrange to supply the `DUAL-EVAL` semantics with a transform (obtained with `MAP-DOWN`) of s, and afterwards we do a reverse transformation (with `MAP-UP`) to obtain the final user-visible state. Futhermore, in making the statement precise we stipulate that the external stimuli, i, to the chip do not enable a hold, a reset, or a test input for the c clock cycles. The fact that n and c are different reflects the fact that a single FM9001 instruction takes several clock cycles to emulate at the `DUAL-EVAL` netlist level. The precise statement of this correctness result is:

```
Theorem.  FM9001=CHIP-SYSTEM-SUMMARY

(let ((h (chip-system$netlist))
      (c (total-microcycles (map-down s) (map-up-inputs i) n)))
  (implies
   (and (fm9001-statep s)
        (no-holds-reset-or-test i c))
   (equal
    (fm9001 s n)
    (map-up
     (simulate-dual-eval-2 'chip-system i (map-down s) h c)))))
```

This theorem is the final event of the file "proofs.events". The function `MAP-DOWN` creates an FM9001 internal state appropriate to begin the execution of instructions. See the file "final-reset.events" for a set of lemmas that prove that such a state can be reached as a result of the reset process.

The following chart is a partial reproduction of a chart in Appendix III of *A Computational Logic Handbook*, by Boyer and Moore [7]. We added the last line. This information was constructed by a static analysis of the final FM9001 library produced by the Boyer-Moore theorem prover.

```
                  Number of lines in understandable statement
                  |     Concept depth of statement
                  |     |     Max concept depth in proof
                  |     |     |     Number of supporters
                  |     |     |     |     Lines of supporters
                  |     |     |     |       |    Depth of proof
                  |     |     |     |       |    |
    85 FM8501   991   157   152   230   2171   18
    86 Goedel   864    48 40414  1741  20002   58
    91 FM9001  1112   120   128  1894  28784   46
```

Appendix III also contains the definitions of these concepts. The “understandable statement” of a theorem is the prettyprinted text of all the definitions used in the theorem except for Boyer-Moore primitives. The number of lines for the understandable statement of the FM9001 proof could be considered to be as high as 9992. The 1112 lines reported above is the sum of the first two lines of the chart just below.

```
                                          Prettyprinted
                                              Lines
    User-level semantics of FM9001             915
    Statement of theorem                       197
    Semantics of HDL                          3459
    FM9001 implementation description         3479
    Existential witness for the clock         1942
                                             -----
    Total:                                    9992
```

To get a rough measure of the size of the FM9001 proof, J Moore modified the theorem prover to count its smallest steps. By adopting a suitable collection of derived inference rules, a formal line-by-line proof using those derived rules can in principle be constructed for each proof step. Among our one-step rules are tautology recognition, instantiation, *modus ponens*, equality rewriting, and cross-multiplication and addition of inequalities.

The number of such proof steps for the FM9001 proof is at least six million. That is, more than six million lines of formal proof were “virtually constructed.” It would be easy to increase this number by reducing the size of our proof steps. For example, the cost of substitution-of-equals-for-equals in HOL [12] is not constant (it is 1 in our count of proof steps) but is proportional to the depth at which the target occurrence is found. Similarly, we recognize very large `IF` expressions as tautologies but charge only one step. We have expressions containing more than 50 `IF` expressions through which there are more than 10^9 branches; however, since the theorem prover prunes the branches dynamically, it did not have to explore all branches.

The total number of primitive proof steps investigated by the theorem prover during its search for the FM9001 proof was more than nineteen million. This means that approximately 32% of all the investigated steps were actually "kept." The term clausifier was called about eighty thousand times. The function that determines the type of a term was called almost fifteen million times and the rewriter was called more than eleven million times.

None of the figures given above for number of lines of statement or proof should be taken too seriously because they were computed prior to our proving a number of additional theorems presented in this report. Furthermore, the counting of lines of formal mathematics is a rather dubious endeavor; thus, for example, in counting the 915 lines of the user-level semantics, we did not count comment lines. Our presentation of such figures, however, does indicate that the theorem proved is a rather large one—it would take almost 200 pages of paper to print it! We believe that the use of computers to check such large theorems is essential because it would be difficult to manipulate such lengthy formulas by hand.

1.3 SUMMARY OF THE PROOF

Users of Nqthm know that there is more to completing a significant proof effort than simply stating the desired theorems and letting the prover prove them. Instead, one needs to prove a number of auxiliary lemmas, and even to define many auxiliary functions. The requirement for such creative effort leads to choices that form a style of proof. In this section we give an overview to the style of definition and proof that we have used in the FM9001 effort.

Our basic approach to the FM9001 specification and verification uses a formalization of an HDL (Section 5.1) based on the four-valued logic introduced above and defined in Chapter 4. The entire implementation netlist for the FM9001 is defined within our HDL. Our specifications are written as Nqthm functions. The goal of the proof effort was to demonstrate that the netlist implementation does indeed satisfy our behavioral specification, as diagrammed in Figure 1.1.

In order to carry out this proof, we made a number of stylistic decisions. For instance, we generally keep functions disabled, especially non-recursive functions, providing enable hints when their definitions are required. Hence we tend to prove rewrite rules about our functions that capture their most important properties. We made a large effort to automate much of the verification of modest-sized HDL modules. We did this by writing a number of Lisp functions and macros that generate appropriate events, with appropriate hints. We also proved lemmas for expanding appropriate calls of the semantic function for our HDL during proofs. We even defined a few Lisp reader macros in order to provide abbreviation mechanisms for constants.

The vast majority of this proof was carried out using a version of Nqthm that has been publicly available since 1987. That system, however, did not contain certain important features, notably the `COND`, `CASE`, and `LET` constructs and the ability to deal with collections of events (including a `DEFTHEORY` event and `ENABLE-THEORY` and `DISABLE-THEORY` hints). Therefore, we actually used the `Pc-Nqthm` "Proof-Checker" interactive enhancement of Nqthm, [16], which includes these features as well as an "interactive" capability. Since we did not take advantage of the latter capability, we were able to replay the events in the more recent Nqthm-1992 system, which has incorporated the features that we needed, without using Pc-Nqthm.

Some parts of the total effort were done rather independently, and yet Nqthm requires a linearly-ordered event list. That is, Nqthm does not directly "understand" the notion of starting with a library, extending it in two different ways, and then merging the results in order to obtain a single library. Hence, suppose we have a library "lib1", which we extend to a library "lib2". Also suppose that separately, we also extend "lib1" to library "lib3". Then typically we merge "lib2" and "lib3" by adding a number of events to the beginning of "lib3" that restores the theorem prover "state" to what it was at the end of "lib1", and then putting the rest of "lib3" after that. That is, we end up with the event sequence obtained by appending together (in this order) "lib1", "lib2", extra enable and disable events, and finally "lib3". Details can be found in the body of this report.

For a description of the official syntax and semantics of the Nqthm-1992 logic, see Chapter 4 of *A Computational Logic Handbook*[7] or the revised version of that chapter which is distributed with the on-line documentation of Nqthm-1992. Nqthm-1992 may be obtained via anonymous FTP from Internet site `ftp.cli.com`.

1.4 ORGANIZATION OF THIS REPORT

In spite of the explanation above about our occasionally ad-hoc ordering of events, we have taken the simple approach of organizing this report around our final sequence of Nqthm events. In order to make the organization somewhat palatable, we have imposed a chapter structure that groups related event files.

Actually, the first chapter after the present one is a bit different: it contains descriptions of all of the Lisp files that we have used in this proof effort, one per section. These files define a number of Lisp utilities that we use to generate Nqthm events. As we explain in the next section, these utilities provide "abbreviations" for Nqthm events but are not strictly necessary. After that, each section corresponds to exactly one Nqthm event file, and the names show the correspondence. For example, Section 5.1 is called "DUAL-EVAL," and corresponds to the file "dual-eval.events". This file may be found in the chapter entitled "The Syntax and Semantics of Our Hardware Description Language," which contains not only the definition of our semantic function `DUAL-EVAL` (in the section mentioned just above) but also contains several related sections, each corresponding to a different Nqthm event file. The next chapter uses a similar naming convention.

Chapter 2

LISP-FILES SUMMARY

This chapter contains a number of Lisp utilities, including ones that transform various types of input into Boyer-Moore event forms.

2.1 SYSDEF

The file "sysdef.lisp" is what we use to build the system. When we load this file into Nqthm-1992, the main things that happen are that a number of Lisp utilities are loaded, as described in Chapter 2, and then the various event files are run, as described in the chapters following Chapter 2. Thus, we load this file when we have made changes and want to check that the sequence of events still runs successfully.

As described in Section 1.3 above, we use some of our Lisp utilities to automate the creation of Nqthm events. For example, much of the control logic in the FM9001 chip is generated (in the form of Nqthm events defining this hardware) from higher-level specifications, using Lisp macros that we have written for this purpose, as described in Section 10.3. Although this has been a useful technique for *obtaining* Nqthm proofs, it is a bit flawed from the point of view of providing *assurance* that appropriate properties of the FM9001 have really been verified. For example, we could have slipped into one of our utilities a redefinition of `PROVE-LEMMA` so that it would always succeed. Or we could have simply executed `(SETQ TRUE FALSE)`. So, in order to obtain an "official" run of our events in the latest (and probably final) release of Nqthm (i.e., Nqthm-1992), we have created a single event file, "fm9001-replay.events", that can be checked in Nqthm with the function `PROVE-FILE`, without any Lisp utilities loaded. This events file is created by loading the file "sysdef.lisp", which (as described above) loads a number of macros, runs the various event files, and then executing the form `(LIBRARY-TO-EVENTS "fm9001")`. The extremely careful reader of this report may wish, in fact, to construct that file in order to see exactly what forms were dynamically computed for submission to Nqthm.

2.2 SYSLOAD

The file "sysload.lisp" is included among the FM9001 files although it is not required to replay the proof. We include it as a simple way to load all of the macro functions described in Chapter 2. If one has run the file "sydef.lisp" to completion, then, as a by-product of the run, a number of Nqthm "libraries" will have been written. It is convenient to resume the Nqthm state for any of these libraries by first loading "sysload.lisp" and then invoking `NOTE-LIB` on the desired library.

2.3 DO-FILES

This file defines a function `DO-FILES` that is used to process a sequence of event files.

See Section 14.3 for the Lisp file "do-files.lisp".

2.4 DO-EVENTS-RECURSIVE

This file defines the function `DO-EVENTS-RECURSIVE` that is used for running a list of events. In particular, this function is used in the file "monotonicity-macros.lisp".

See Section 14.4 for the Lisp file "do-events-recursive.lisp".

2.5 DISABLE

The general problem solved by these macros is to merge different event files. This file contains macros used in files "math-enable.events" and "math-disable.events" (see Sections 13.1 and 3.4) for controlling the events that are enabled at a given point. (The Nqthm-1992 command `SET-STATUS` was not available at the time these macros were written, or else they probably would not have been necessary.) If Nqthm allowed partially-ordered theories, then perhaps this sort of problem would disappear.

In particular, the file "math-enable.events" contains the following forms.

```
(better-disable-back-to b-knownp)

(enable-theory math-theory)
```

The event file to be run immediately after "math-enable.events" is the file "alu-interpretation.events", which was developed within the environment that existed just before the event `B-KNOWNP` in "approx.events". The first form above disables a number of events so that the enabled events will closely approximate those that were enabled at the time the event B-KNOWNP was submitted. Then the second form enables all events in a certain theory, MATH-THEORY. The expectation is that the resulting environment will sufficiently approximate the one in which "alu-interpretation.events" was created so that the events in that file will run successfully.

Incidentally, the aforementioned theory MATH-THEORY was created much earlier in the event sequence, in the file "math-disable.events", by the use of the macro DISABLE-BACK-TO. In essence, that macro "saved" the current enabled state into the theory MATH-THEORY, for later use as discussed above.

See Section 14.5 for the Lisp file "disable.lisp".

2.6 MACROS

This file contains a number of macros used for generating events related to hardware modules. Some are summarized below.

When we represent a hardware module in the Nqthm logic, we produce several Nqthm events in a stylized manner, including the following: a module recognizer, a module generator and netlist builder, an Nqthm specification function for the module, and one or more correctness theorems for the module. In addition, some of these events are to be disabled. The macros presented below are used to reduce the effort required to construct all of these events.

First let us examine in some detail a macro that takes the name of a "hardware" specification function and produces a series of Nqthm events that it submits to the system. For example, consider the specification of a "mode" recognizer given below.

```
(defn decode-mode (op0 op1 op2 op3)
  (b-nor (b-nor3 op0 op1 op2)
         op3))
```

Given that the function DECODE-MODE has been defined, the macro call (DEFN-TO--MODULE DECODE-MODE) results in the creation of the Nqthm event forms as shown below. Note that the formal parameters of DECODE-MODE are each intended to be Boolean signal values, and that this function produces a single Boolean output. DEFN-TO-MODULE requires this "shape"; it cannot be used, for example, to produce modules with multiple outputs.

Here are the events produced by evaluating (DEFN-TO-MODULE DECODE-MODE). In each case below we precede the event form with a short comment.

Define the four-valued logic specification function for this module.

```
(DEFN F$DECODE-MODE (OP0 OP1 OP2 OP3)
```

```
  (F-NOR (F-NOR3 OP0 OP1 OP2)
         OP3))
```

The specification above is to be kept disabled.

```
(DISABLE F$DECODE-MODE)
```

Define the module constant.

```
(DEFN DECODE-MODE* ()
  '(DECODE-MODE (OP0 OP1 OP2 OP3)
                (W-1)
                ((G-0 (W-0) B-NOR3 (OP0 OP1 OP2))
                 (G-1 (W-1) B-NOR (W-0 OP3)))
                NIL))
```

Define a recognizer for netlists that correctly represent this module, in terms of the module constant just defined as well as the previously defined netlist recognizers for its submodules.

```
(DEFN DECODE-MODE& (NETLIST)
  (AND (EQUAL (LOOKUP-MODULE 'DECODE-MODE NETLIST)
              (DECODE-MODE*))
       (B-NOR3& (DELETE-MODULE 'DECODE-MODE NETLIST))
       (B-NOR& (DELETE-MODULE 'DECODE-MODE NETLIST))))
```

The netlist recognizer is to be kept disabled.

```
(DISABLE DECODE-MODE&)
```

Define the netlist constant for this module so that it includes the module as well as all the netlist constants for its submodules.

```
(DEFN DECODE-MODE$NETLIST ()
  (CONS (DECODE-MODE*)
        (UNION (B-NOR3$NETLIST)
               (B-NOR$NETLIST))))
```

Prove a correctness theorem for the module. It says that if `NETLIST` correctly represents the module, then the output vector from this module is equal to the one-bit vector containing the value given by the four-valued specification function. Note that an analogous lemma would also have been proved about the new state of the module, if the module had been a state-holding module.

```
(PROVE-LEMMA DECODE-MODE$VALUE (REWRITE)
  (IMPLIES (DECODE-MODE& NETLIST)
           (EQUAL (DUAL-EVAL 0 'DECODE-MODE
                             (LIST OP0 OP1 OP2 OP3)
                             STATE NETLIST)
                  (LIST (F$DECODE-MODE OP0 OP1 OP2 OP3))))
  ((ENABLE DECODE-MODE& F$DECODE-MODE B-NOR3$VALUE B-NOR$VALUE)
   (DISABLE-THEORY F-GATES)))
```

The `VALUE` lemma above is to be kept disabled.

```
(DISABLE DECODE-MODE$VALUE)
```

The final lemma simply equates the four-value specification function with the original (input, Boolean) specification function in the case that the inputs are all Boolean.

```
(PROVE-LEMMA F$DECODE-MODE=DECODE-MODE (REWRITE)
  (IMPLIES (AND (BOOLP OP0)
                (BOOLP OP1)
                (BOOLP OP2)
                (BOOLP OP3))
           (EQUAL (F$DECODE-MODE OP0 OP1 OP2 OP3)
                  (DECODE-MODE OP0 OP1 OP2 OP3)))
  ((ENABLE BOOLP-B-GATES F$DECODE-MODE DECODE-MODE)
   (DISABLE-THEORY F-GATES B-GATES)))
```

Let us turn to brief summaries of utilities provided in this file.
`(DEFN-TO-MODULE spec &KEY value-lemma boolp-value-lemma)` has been described above. It converts simple Nqthm specifications to Nqthm events, including definitions of generators and recognizers for modules and netlists, as well as appropriate theorems. The specification argument `spec` must be the name of an Nqthm `DEFN` event whose formal parameters are each intended to be a signal, where the function's value is intended to be the module's single output value.

`(MODULE-PREDICATE generator)` writes the netlist predicate for simple modules. For example, if the module is defined by `(M*)`, then `(MODULE-PREDICATE M*)` creates `(M& NETLIST)`.

`(MODULE-NETLIST generator)` assembles the netlist for simple modules. For example, if the module is defined by `(M*)`, then `(MODULE-NETLIST M*)` creates `(M$NETLIST NETLIST)`.

`(DESTRUCTURING-LEMMA generator)` creates a lemma that explicitly states how to destructure a module definition. Because of quirks in equality reasoning, it is not effective simply to let module definitions open up. It also creates two `DISABLE` events. For example, after defining the n-bit-wide adder (BV-ADDER* n), the following macro call creates the events shown below it.

```
(destructuring-lemma bv-adder*)

(PROVE-LEMMA BV-ADDER*$DESTRUCTURE (REWRITE)
  (AND (EQUAL (CAR (BV-ADDER* N))
              (INDEX 'BV-ADDER N))
       (EQUAL (CADR (BV-ADDER* N))
              (CONS (INDEX 'CARRY 0)
                    (APPEND (INDICES 'A 0 N)
                            (INDICES 'B 0 N))))
       (EQUAL (CADDR (BV-ADDER* N))
              (APPEND (INDICES 'SUM 0 N)
                      (LIST (INDEX 'CARRY N))))
       (EQUAL (CADDDR (BV-ADDER* N))
              (BV-ADDER-BODY 0 N))
       (EQUAL (CADDDDR (BV-ADDER* N)) NIL)))

(DISABLE BV-ADDER*)

(DISABLE BV-ADDER*$DESTRUCTURE))
```

The macro call (MODULE-GENERATOR (generator . args) name inputs outputs body state) is used to define a module. It is followed by the events created by the macro DESTRUCTURING-LEMMA. (GENERATE-BODY-INDUCTION-SCHEME generator) creates a (DUAL-EVAL 1 ...) induction scheme for body generators of the form (IF test something (CONS occurrence recursive-call)).

The file concludes with definitions of a couple of Lisp reader macros. First, a "#v" reader macro is an abbreviation mechanism for reading in bit-vector constants. For example, #v001 is read as (LIST T F F). Second, a reader macro for index names for DUAL-EVAL netlists is defined. For example, #i(A 10) is read as (INDEX 'A 10), and #i(A 0 10) is read as (INDICES 'A 0 10).

See Section 14.6 for the Lisp file "macros.lisp".

2.7 EXPAND

The EXPAND macro is used by other utilities to generate events. What (EXPAND form) returns is the term that the Nqthm simplifier is expected to produce when trying to prove a lemma about form. For example, suppose that we submit the following form.

```
(expand '(plus (add1 x) y))
```

In that case, Nqthm returns the following answer.

```
(ADD1 (PLUS X Y))
```

The macro `EXPAND-LEMMA` is similar to `EXPAND`, but it produces a `PROVE-LEMMA` event that equates the input term with the result produced by `EXPAND`. For example, the input immediately below causes the `PROVE-LEMMA` event below it to be submitted to the theorem prover.

```
(expand-lemma plus-add1-first-arg (rewrite)
              nil
              (plus (add1 x) y))

(PROVE-LEMMA PLUS-ADD1-FIRST-ARG (REWRITE)
  (EQUAL (PLUS (ADD1 X) Y)
         (ADD1 (PLUS X Y))) NIL)
```

See Section 14.7 for the Lisp file "expand.lisp".

2.8 VECTOR-MACROS

This file defines a single Lisp macro, `VECTOR-MODULE`. It is similar in spirit to the macro `DEFN-TO-MODULE` discussed earlier in Section 2.6, in that it generates Nqthm events from a higher-level specification of a module. Perhaps the simplest way to explain what it does is to show how this macro works on an example. Suppose that we have defined a module `B-BUF` that buffers a single input line into a single output line, and we want to create a vector version that has an arbitrary number n of input lines and has n output lines, by connecting n `B-BUF` modules in parallel. For example, for $n = 3$ we might want to obtain something like the following module.

```
'(V-BUF_3 (A_0 A_1 A_2)
          (Y_0 Y_1 Y_2)
          ((G_0 (Y_0) B-BUF (A_0))
           (G_1 (Y_1) B-BUF (A_1))
           (G_2 (Y_2) B-BUF (A_2)))
          NIL)
```

The following call of the macro `VECTOR-MODULE` creates the module shown above. More precisely, if we apply the function `LISP-NETLIST` (see Section 5.9) to the module created by the call of `VECTOR-MODULE` shown below then we obtain the module displayed above. Here is how this macro call expands; a quick glance should give a rough idea of what is going on, so we eschew further comments.

```
(macroexpand
 '(VECTOR-MODULE v-buf    ;the name of a parametrized module
                 (g (y) b-buf (a))
                 ((v-threefix a))
                 :enable (f-buf)))

(PROGN

 (DEFN V-BUF$BODY (M N)
   (IF (ZEROP N) NIL
       (CONS (LIST (INDEX 'G M) (LIST (INDEX 'Y M)) 'B-BUF
                   (LIST (INDEX 'A M)))
             (V-BUF$BODY (ADD1 M) (SUB1 N)))))

 (DISABLE V-BUF$BODY)

 (MODULE-GENERATOR (V-BUF* N) (INDEX 'V-BUF N) (INDICES 'A 0 N)
                   (INDICES 'Y 0 N) (V-BUF$BODY 0 N) NIL)

 (DEFN V-BUF& (NETLIST N)
   (AND (EQUAL (LOOKUP-MODULE (INDEX 'V-BUF N) NETLIST)
               (V-BUF* N))
        (B-BUF& (DELETE-MODULE (INDEX 'V-BUF N) NETLIST))))

 (DISABLE V-BUF&)

 (DEFN V-BUF$NETLIST (N) (CONS (V-BUF* N) (B-BUF$NETLIST)))

 (PROVE-LEMMA V-BUF$UNBOUND-IN-BODY (REWRITE)
   (IMPLIES (LESSP L M)
            (UNBOUND-IN-BODY (INDEX 'Y L) (V-BUF$BODY M N)))
   ((ENABLE V-BUF$BODY UNBOUND-IN-BODY)))

 (DISABLE V-BUF$UNBOUND-IN-BODY)

 (PROVE-LEMMA V-BUF$BODY-VALUE (REWRITE)
   (IMPLIES (AND (B-BUF& NETLIST) (EQUAL BODY (V-BUF$BODY M N)))
```

```
          (EQUAL (COLLECT-VALUE (INDICES 'Y M N)
                                (DUAL-EVAL 1 BODY BINDINGS
                                           STATE-BINDINGS
                                           NETLIST))
                 (V-THREEFIX
                  (COLLECT-VALUE (INDICES 'A M N) BINDINGS))))
  ((INDUCT (VECTOR-MODULE-INDUCTION BODY M N BINDINGS
                                    STATE-BINDINGS NETLIST))
   (DISABLE-THEORY F-GATES)
   (ENABLE V-BUF$BODY B-BUF$VALUE V-BUF$UNBOUND-IN-BODY V-THREEFIX
           F-BUF)))

(DISABLE V-BUF$BODY-VALUE)

(PROVE-LEMMA V-BUF$VALUE (REWRITE)
  (IMPLIES (AND (V-BUF& NETLIST N)
                (AND (PROPERP A) (EQUAL (LENGTH A) N)))
           (EQUAL (DUAL-EVAL 0 (INDEX 'V-BUF N) A STATE NETLIST)
                  (V-THREEFIX A)))
  ((ENABLE V-BUF& V-BUF$BODY-VALUE V-BUF*$DESTRUCTURE)))

(DISABLE V-BUF$VALUE)

)
```

See Section 14.8 for the Lisp file "vector-macros.lisp".

2.9 PRIMP-DATABASE

This file defines the Lisp constant `COMMON-LISP-PRIMP-DATABASE`, which is used in some Lisp macros as well as in the definition of the Nqthm constant (i.e., 0-ary function) `PRIMP-DATABASE`. This constant is a list that represents the primitive modules in our hardware description language. Each entry in this list may be viewed as a record whose fields describe properties of a given primitive module.

Let us examine one such entry: the entry for the Boolean two-input *and* gate. We include a comment in front of each field.

```
(b-and

 ;;  These are the slope and intercept delays for LSI Logic's
 ;;  implementation of this module.
 (delays        ((144 422) (54 707)))
```

```
;;  The module can drive 10 standard loads.  This is used for
;;  crude estimates of fan-out restrictions.  The more accurate
;;  estimates are given by the  delays field shown just above.
(drives         10)

;;  Each of the inputs is expected to be Boolean.
(input-types    boolp boolp)

;;  The inputs are named  a and  b.
(inputs         a b)

;;  The standard input loads are both 1.
(loadings       1 1)

;;  The name of the corresponding LSI Logic model is  an2.
(lsi-name       . an2)

;;  The output(s) combinationally depend on both inputs.
(out-depends    (a b))

;;  There is a single Boolean output.
(output-types   boolp)

;;  The output name is  z.
(outputs        z)

;;  The vector of outputs from this device contains the single
;;  value obtained by ``evaluating'' the indicated Nqthm term.
(results        . (cons (f-and a b) 'nil))

;;  The remaining fields are intended to approximately measure
;;  the ``cost'' of the primitive.
(gates          . 2)
(primitives     . 1)
(transistors    . 6))
```

For state-holding modules we have some additional fields. For example, the basic single-bit D flip-flop device FD1 has the following additional fields in its entry.

```
;;  The new state is obtained by ``evaluating'' the indicated
```

```
;;  Nqthm term.
(new-states    . (f-buf d))

;;  The state is a single Boolean value.
(state-types   . boolp)

;;  The name of the state is  state.
(states        . state)
```

See Section 14.9 for the Lisp file "primp-database.lisp".

2.10 PRIMITIVES

This file contains functions and macros to automate the creation of `DUAL-EVAL` lemmas for the primitives. For each primitive listed in our primitive database (see Section 2.9), a series of lemmas is proven that describes the effect of the `DUAL-EVAL` simulator on each language primitive.

See Section 14.10 for the Lisp file "primitives.lisp".

2.11 CONTROL-LISP

This file defines a number of macros that are used to define the control logic in Section 10.3. We defer explanation of these macros until that section.

See Section 14.11 for the Lisp file "control.lisp".

2.12 EXPAND-FM9001

This file defines a number of macros that write lemmas about the transitions between the major FM9001 control states. There are also some definitions used to compute the amount of time required for the various instruction types. These macros are used by the event file "expand-fm9001.events" (Section 11.1).

See Section 14.12 for the Lisp file "expand-fm9001.lisp".

2.13 MONOTONICITY-MACROS

The Lisp utilities in this file are used in the event file "approx.events"; see Section 11.3 for further discussion.

See Section 14.13 for the Lisp file "monotonicity-macros.lisp".

2.14 TRANSLATE

This file contains a Common Lisp program that translates netlists from the `DUAL--EVAL` hardware description language into NDL, LSI Logic's Network Description Language. We used this program to produce the FM9001 netlist which was delivered to LSI Logic and subsequently used by them to manufacture working FM9001 microprocessors. The FM9001 netlist is presented in Chapter 15. The initial comment in the file "translate.lisp" describes how to print netlists, including the one used for the FM9001.

The Common Lisp code in this file has not been subjected to any form of verification with Nqthm-1992. However, the printing process is a very straightforward translation and occupies only about two pages of text. To validate the correctness of our translation process, we simulated the converted netlist using LSI Logic's simulator `lsim` and checked to see that `lsim` produced the same answers using a variety of test vectors.

See Section 14.14 for the Lisp file "translate.lisp".

2.15 PURIFY

This file defines the Lisp utility `LIBRARY-TO-EVENTS`, which converts an Nqthm or Pc-Nqthm library that may have been created in an environment containing many "unofficial" Lisp macros or even redefinitions of Nqthm functions into a single Nqthm event file whose syntax is acceptable to the `PROVE-FILE` function of Nqthm-1992. Since `PROVE-FILE` is very careful in what it accepts as valid Nqthm input, this approach gives us added confidence in the theorems that we claim to have proved.

The exact approach is as follows.

1. Start up Nqthm and submit the form
   `(LOAD "sysdef.lisp")`
   — this creates the "proofs" library.

2. Submit the form
   `(library-to-events "fm9001")`.

The file `fm9001-replay.events` is thereby created. To run this single file one may then submit the form:

```
(prove-file-out "fm9001-replay")
```

Chapter 3

SOME ARITHMETIC LIBRARIES

3.1 BAGS

This file defines the `BAGS` theory, which provides rewrite rules about bags, i.e., multi-sets. Bags are represented by lists and are manipulated by the following five functions: `DELETE`, `BAGDIFF`, `BAGINT`, `OCCURRENCES`, and `SUBBAGP`. A theory `BAGS` is defined that contains a number of rewrite rules about these functions.

See Section 14.16 for the event file "bags.events".

3.2 NATURALS

This file defines several theories for reasoning about natural numbers, which are gathered together into a single `NATURALS` theory at the end of the file.

```
(deftheory naturals
  (addition
   multiplication
   remainders
   quotients
   exponentiation
   logs
   gcds))
```

Here is a brief description of the events in this file, adapted from Bevier's hardware library [3]. Some of the ideas are analogous to ideas in the integers library, which are given a more thorough explanation below.

The first part of the development of our arithmetic library is devoted to rules which canonicalize terms involving PLUS and DIFFERENCE. Rewrite rules canonicalize terms involving these two functions into "preferable" forms. For instance, DIFF-DIFF-ARG1 canonicalizes a DIFFERENCE expression whose first argument is a DIFFERENCE.

Lemma. DIFF-DIFF-ARG1

```
(equal (difference (difference x y) z)
       (difference x (plus y z))))
```

The lemmas about TIMES, REMAINDER and QUOTIENT have been distilled from Boyer and Moore's 1979 book [5], together with the proof of FM8501 [13], [14] and the proof of Kit [2], [4].

See Section 14.17 for the event file "naturals.events".

3.3 INTEGERS

This file defines the INTEGERS theory for reasoning about integers. The following functions are defined: INTEGERP, FIX-INT, IZEROP, ILESSP, ILEQ, IPLUS, INEG, IDIFFERENCE, IABS, ITIMES, IQUOTIENT, IREMAINDER, IDIV, IMOD, IQUO, IREM, and SUBSETP.

The rest of this section is taken largely from [17] and explains some features of this library, especially its metalemmas. This library has evolved over the years, starting with the test suite for the Boyer-Moore prover, continuing to evolve in the dissertations of Hunt [13] and Bevier [2], and reaching the present form (at least approximately) with Kaufmann [17]. It has been designed so that one can use it profitably without reading further. We do, however, present some ideas on how better to take advantage of the library. For example, it is suggested that one may want to experiment by enabling the rewrite rule ILESSP-TRICHOTOMY.

The first section below briefly summarizes the basic integer functions and some rewrite rules defined in this library. The second section states the metalemmas and describes the algorithms they implement, and suggests possible actions on the part of the user to adjust the set of metalemmas that are enabled. The final section has some miscellaneous caveats and suggestions.

3.3.1 Basics

The basic functions defined in this library are defined in terms of natural number functions, and have been put into the following theory.

```
(deftheory all-integer-defns
  (integerp fix-int izerop ilessp ileq iplus ineg idifference
```

```
            iabs itimes iquotient iremainder idiv imod iquo irem))
```

In accordance with the usual way arithmetic functions are defined in Nqthm, these functions act as though non-integer arguments are coerced to `0`. The history has been left in a state where integer functions (e.g., `ILESSP`, `FIX-INT`, `IPLUS`, ...) have been disabled, with three exceptions: `IZEROP`, `ILEQ`, and `IDIFFERENCE`. Therefore, there are very few rules about `IZEROP` (but plenty of rules about `FIX-INT`), few rules about `ILEQ` (but many rules about `ILESSP`), and few rules about `IDIFFERENCE` (but many about `IPLUS` and `INEG`). If one wants to prove a basic integer fact by opening up functions that are normally disabled, an `(ENABLE-THEORY INTEGER-DEFNS)` hint should be given, where `INTEGER-DEFNS` is defined as follows.

```
(deftheory integer-defns
  (integerp fix-int ilessp iplus ineg iabs itimes
            iquotient iremainder idiv imod iquo irem))
```

Let us turn briefly to rewrite rules. There are rewrite rules expressing the commutativity and associativity of `IPLUS` and `ITIMES`. (Notice that for the metalemmas, it suffices to deal with terms that are right-associated with respect to these operations.) Also included are distributive laws that push `IPLUS` out of `ITIMES` expressions and out of `INEG` expressions and that push `INEG` out of `ITIMES` expressions. Rules have been proved which state that `0` and `1` are the respective identities for integer addition (i.e., `IPLUS`) and integer multiplication (i.e., `ITIMES`). There are also a number of trivial "type" rules, e.g., `IPLUS` always returns an integer and implicitly applies `FIX-INT` to its arguments. Finally, there are rules for combining constants in `IPLUS` terms, so that for example `(IPLUS 3 (IPLUS X 7))` is rewritten to `(IPLUS 10 X)`, `(EQUAL (IPLUS 10 X) (IPLUS Z 3))`, which is then rewritten to `(EQUAL (IPLUS 7 X) (FIX-INT Z))`, and finally `(ILESSP (IPLUS 3 Z) (IPLUS 10 X))` is rewritten to `(NOT (ILESSP (IPLUS 6 X) Z))`. Here are the lemmas used (besides commutativity) in these respective examples:

```
Theorem. IPLUS-CONSTANTS
(equal (iplus (add1 i) (iplus (add1 j) x))
       (iplus (plus (add1 i) (add1 j)) x))
```

```
Theorem. CANCEL-CONSTANTS-EQUAL
(equal (equal (iplus (add1 i) x)
              (iplus (add1 j) y))
       (if (lessp i j)
           (equal (fix-int x)
                  (iplus (difference j i) y))
           (equal (iplus (difference i j) x)
                  (fix-int y))))

Theorem. CANCEL-CONSTANTS-ILESSP
(equal (ilessp (iplus (add1 i) x)
               (iplus (add1 j) y))
       (if (lessp i j)
           (not (ilessp (iplus (sub1 (difference j i)) y)
                        x))
           (ilessp (iplus (difference i j) x)
                   y)))
```

It had seemed that metalemmas might be needed for dealing with such constants, but in fact numeric constants are lower in the Nqthm term order than terms with variables, so commutativity and associativity of `IPLUS` tend to push constants to the front of the sum.

3.3.2 Metalemmas

The reader is referred to the events list for the definitions of the relevant meta-functions. Here we simply state the metalemmas and make a few comments about them.

The first metalemma cancels pairs `V` and `(INEG V)` in a given sum. The idea of `CANCEL-INEG` is that it when it sees a sum `(IPLUS x y)`, it looks to see if the formal negative of `x` appears as a summand of `y`, where by the "formal negative" of `x` we mean `(INEG x)` unless `x` is of the form `(INEG x1)`, in which case we mean `x1`. If so, then `CANCEL-INEG` cancels that pair.

```
Theorem. CORRECTNESS-OF-CANCEL-INEG
(equal (eval$ t x a)
       (eval$ t (cancel-ineg x) a)))
```

The next two metalemmas are for cancelling common summands on both sides of an equality or inequality (respectively). There are no particular surprises here. Notice that since the function symbol `ILEQ` is kept enabled, there is no need for a corresponding cancellation lemma for `ILEQ`.

```
Theorem. CORRECTNESS-OF-CANCEL-IPLUS
(equal (eval$ t x a)
       (eval$ t (cancel-iplus x) a))

Theorem. CORRECTNESS-OF-CANCEL-IPLUS-ILESSP
(equal (eval$ t x a)
       (eval$ t (cancel-iplus-ilessp x) a))
```

The next two metalemmas are useful for automatically cancelling factors on both sides of an equation or inequality, respectively. The factors must be explicit, however. So for example, the first of these lemmas below will simplify the expression

```
(equal (itimes x y) (itimes x z))
```

to

```
(if (equal (fix-int x) 0)
    t
  (equal (fix-int y) (fix-int z)))
```

But it will not simplify the expression `(EQUAL (IPLUS (ITIMES X Y) X) (IPLUS X (ITIMES X Z)))`. Therefore, there are stronger versions of these metalemmas, namely the third and fourth lemmas below (which strengthen the first and second, respectively). Because these last two are stronger, the first two are actually disabled in this library. If one finds these last two to be a nuisance for some reason, however (e.g., if they are too slow), it is of course easy enough to disable the last two and enable the first two. It seems so far that perhaps very little speed is sacrificed by having the latter pair enabled instead of the former.

```
Theorem. CORRECTNESS-OF-CANCEL-ITIMES (disabled)
(equal (eval$ t x a)
       (eval$ t (cancel-itimes x) a))

Theorem. CORRECTNESS-OF-CANCEL-ITIMES-ILESSP (disabled)
(equal (eval$ t x a)
       (eval$ t (cancel-itimes-ilessp x) a))

Theorem. CORRECTNESS-OF-CANCEL-ITIMES-FACTORS
(equal (eval$ t x a)
       (eval$ t (cancel-itimes-factors-expanded x) a))

Theorem. CORRECTNESS-OF-CANCEL-ITIMES-ILESSP-FACTORS
(equal (eval$ t x a)
```

```
       (eval$ t (cancel-itimes-ilessp-factors x) a))
```

Remark. Notice that the metalemma CORRECTNESS-OF-CANCEL-ITIMES-FACTORS refers to the metafunction CANCEL-ITIMES-FACTORS-EXPANDED. There is in fact a function CANCEL-ITIMES-FACTORS, but it has seemed possible to gain efficiency by expanding all occurrences of AND and OR in the body of this function, since the executable counterparts of AND and OR are not "lazy" in Nqthm. (Preliminary tests suggests only slight gains in efficiency, however.) The Lisp macro NQTHM-MACROEXPAND, whose definition appears in a comment in the events below, was used to create the expanded version CANCEL-ITIMES-FACTORS-EXPANDED automatically. Some other metafunction definitions have been similarly created. The following rule, followed by (DISABLE CANCEL-ITIMES-FACTORS-EXPANDED), eliminates the need to reason about the "expanded" function CANCEL-ITIMES--FACTORS-EXPANDED:

```
Theorem. CANCEL-ITIMES-FACTORS-EXPANDED-CANCEL-ITIMES-FACTORS
(equal (cancel-itimes-factors-expanded x)
       (cancel-itimes-factors x))
```

Our final group of metalemmas takes care of some details not handled by the metalemmas above. The first of these, CORRECTNESS-OF-CANCEL-FACTORS-0, replaces a term that equates a product with 0 by the disjunction of the assertions saying that some factor equals 0. (More precisely, each disjunct says that FIX-INT applied to the factor equals 0.) The product can in fact be implicit, for as the following example shows, the function CANCEL-FACTORS-0 implicitly factors the expression completely.

```
(cancel-factors-0
 '(equal '0 (itimes z (iplus x (itimes x y)))))

  =

'(OR (EQUAL (FIX-INT Z) '0)
     (OR (EQUAL (FIX-INT (IPLUS '1 Y)) '0)
         (EQUAL (FIX-INT X) '0)))
```

Here then are the metalemmas that apply when a product is set equal to (or less than, or greater than) 0.

```
Theorem. CORRECTNESS-OF-CANCEL-FACTORS-0
```

```
(equal (eval$ t x a)
       (eval$ t (cancel-factors-0 x) a))
```

```
Theorem. CORRECTNESS-OF-CANCEL-FACTORS-ILESSP-0
(equal (eval$ t x a)
       (eval$ t (cancel-factors-ilessp-0 x) a))
```

Finally, there are metalemmas for removing `INEG` terms from equations and inequalities. The desirability of performing such normalizations is illustrated by the following example:

```
(implies (equal (iplus x y)
                (iplus u v))
         (equal (fix-int x)
                (iplus u (idifference v y))))
```

Without the first metalemma below, the integer library fails to prove this obvious fact. However, the system may use that metalemma by replacing the term

```
(equal (fix-int x)
       (iplus u (iplus v (ineg y))))
```

with the term

```
(if (integerp (fix-int x))
    (equal (iplus y (fix-int x))
           (iplus u v))
    f)
```

which in turn simplifies to the hypothesis of the original implication. Here then are those final two metalemmas.

```
Theorem. CORRECTNESS-OF-CANCEL-INEG-TERMS-FROM-EQUALITY
(equal (eval$ t x a)
       (eval$ t (cancel-ineg-terms-from-equality-expanded x) a))
```

```
Theorem. CORRECTNESS-OF-CANCEL-INEG-TERMS-FROM-INEQUALITY
(equal (eval$ t x a)
       (eval$ t (cancel-ineg-terms-from-inequality-expanded x) a))
```

These last two metalemmas do *not* subsume the first one, CORRECTNESS-OF--CANCEL-INEG. For, these lemmas only apply to INEG terms that occur as a summand of the left or right side of an equation or inequality, while CORRECTNESS-OF--CANCEL-INEG cancels negatives in arbitrary sums.

3.3.3 Caveats

Here are a few things to watch out for when using this library. There are probably many others.

First of all, a previous version of the integers library had more power for reasoning about terms in which integer functions are applied to arguments that are not known to be integers. For example, the following rewrite rule IPLUS-LEFT-ID is now disabled.

```
(implies (not (integerp x))
         (equal (iplus x y)
                (fix-int y)))
```

Here is a list of other such rules about non-integers that have been left disabled: INEG-OF-NON-INTEGERP, IPLUS-RIGHT-ID, NOT-INTEGERP-IMPLIES-NOT-EQUAL--IPLUS, IPLUS-LEFT-ID, NOT-INTEGERP-IMPLIES-NOT-EQUAL-ITIMES. These rules have been disabled because one can pay a rather severe performance penalty otherwise. The example lemma FOUR-SQ shown below created a time triple of [0.1 62.4 0.2] on a Sun3/60 with the integer library loaded; but when the above rules were first enabled, the time triple was instead [0.1 84.9 0.2]. Two related rules that have also been left disabled to avoid backchaining are ITIMES-ZERO1 and ITIMES-ZERO2. Here is the former of those:

```
(implies (equal (fix-int x) 0)
         (equal (itimes x y) 0))
```

With those two rules enabled in addition to the five listed above, the proof of this example took time [0.1 797.6 0.3]! These rules are not too important anyhow, given the metalemma CORRECTNESS-OF-CANCEL-FACTORS-0. A more difficult case is the following rewrite rule, named SAME-FIX-INT-IMPLIES-NOT-ILESSP.

```
(implies (equal (fix-int x) (fix-int y))
         (not (ilessp x y)))
```

Perhaps it would be better to leave this rule enabled, but it has been disabled. Another such potentially useful rewrite rule that is kept disabled is the following, named `IZEROP-ILESSP-0-RELATIONSHIP`.

```
(equal (equal (fix-int x) 0)
       (and (not (ilessp x 0))
            (not (ilessp 0 x))))
```

The following rewrite rule `ILESSP-TRICHOTOMY` is kept disabled, though it is sometimes very useful to enable this rule.

```
(implies (not (ilessp x y))
         (equal (ilessp y x)
                (not (equal (fix-int x) (fix-int y)))))
```

If both `IZEROP-ILESSP-0-RELATIONSHIP` and `ILESSP-TRICHOTOMY` are enabled, one is in extreme danger of entering an infinite loop in the rewriter! In fact there is a note in the event list below suggesting that this rule caused trouble in the proof of `CORRECTNESS-OF-CANCEL-ITIMES-ILESSP-FACTORS`.

Here is the test problem mentioned above, which was suggested by William Pase of Odyssey Research Associates.

```
Definition.
(square x) =
(itimes x x)

Definition.
(four-squares a b c d) =
(iplus (square a)
       (iplus (square b)
              (iplus (square c)
                     (square d))))

Theorem. FOUR-SQ
(equal (itimes (four-squares a b c d)
               (four-squares r s t0 u))
       (four-squares
        (iplus (itimes a r)
               (iplus (itimes b s)
                      (iplus (itimes c t0)
                             (itimes d u))))
```

```
(iplus (itimes a s)
       (iplus (ineg (itimes b r))
              (iplus (itimes c u)
                     (ineg (itimes d t0)))))
(iplus (itimes a t0)
       (iplus (ineg (itimes b u))
              (iplus (ineg (itimes c r))
                     (itimes d s))))
(iplus (itimes a u)
       (iplus (itimes b t0)
              (iplus (ineg (itimes c s))
                     (ineg (itimes d r)))))))
```

See Section 14.18 for the event file "integers.events".

3.4 MATH-DISABLE

This file disables appropriate lemmas. See Section 2.5 for more details.
See Section 14.19 for the file "math-disable.events".

Chapter 4

SOME FUNDAMENTAL NOTIONS

This chapter introduces some basic lemmas and definitions that are used throughout this work, including some rewrite rules for numbers and lists, the `INDEX` shell for creating indexed names, lookup functions, basic Boolean and four-valued logic functions, constructors for manipulating the machine's memory, and some basic tree-oriented functions.

This chapter also introduces our four-valued logic, which has the values `T` (Boolean "true"), `F` (Boolean "false"), `(Z)` (a high impedance, intermediate value), and `(X)` (unknown). The function `FOURP` is a recognizer for these four values, i.e., it returns `T` on each of the four values above and `F` on everything else. The function `FOURFIX` returns its argument unchanged if its argument is a four-valued logic value (i.e., a `FOURP`), and otherwise returns `(X)`; in other words, `FOURFIX` can be viewed as a coercion function that coerces all Nqthm objects that are not four-valued logic values to the value `(X)`, i.e., unknown. There are analogous functions `THREEP` and `THREEFIX` for a three-valued logic that excludes `(Z)`, as well as `BOOLP` and `BOOLFIX` for ordinary two-valued Boolean logic based on `T` and `F`.

4.1 INTRO

The file "intro.events" contains a number of basic definitions and lemmas for arithmetic, lists, and trees.

First we define functions `IF*`, `OR*`, `AND*`, and `NOT*`, which are really the same as their "unstarred" counterparts, but give us finer control over the theorem prover when we need it. The problem with the built-in propositional functions is that the theorem prover has some heuristics for them that we sometimes prefer to override. Along the same lines, we prove some trivial lemmas such as `CAR-CDR-IF-CONS` that help the theorem prover in its treatment of `IF` expressions.

The arithmetic lemmas that come next are in some cases subsumed by the natural number library (see Section 3.2), but these were proved in isolation before we incorporated those libraries into our work. For example, the following lemma tells the Boyer-Moore prover to use the fact that the sum of two natural numbers is zero if and only if both of those numbers are zero.

```
(equal (equal (plus a b) 0)
       (and (zerop a) (zerop b)))
```

The files also define a number of list and tree processing functions, along with rewrite rules that support their use. For instance, we define the following function that returns all but the first `N` elements of a given list `L`.

Definition.

```
(restn n l)
 =
(if (listp l)
    (if (zerop n)
        l
      (restn (sub1 n) (cdr l)))
  l)
```

Here is an example of a rewrite rule about lists. It simplifies the application of `RESTN` to a list that is formed by the built-in list concatenation function, `APPEND`.

```
(equal (restn n (append a b))
       (append (restn n a)
               (restn (difference n (length a)) b)))
```

See Section 14.20 for the event file "intro.events".

4.2 LIST-REWRITES

The lemmas in this file were useful for very technical reasons, probably not worth going into here beyond the comments in the file "list-rewrites.events".

See Section 14.21 for the event file "list-rewrites.events".

4.3 INDICES

These events define the `INDEX` shell constructor to help us create `DUAL-EVAL` netlists. For example, the term `(INDEX 'G 3)` is a legitimate Nqthm term, which we think of intuitively as G_3. In fact, the index reader (see Section 2.6) tells the Lisp reader to read

```
#i(A 3)
```

exactly as though it were

```
(INDEX 'A 3)
```

The function `INDICES` is also defined here. It can be used to create a list of `INDEX` terms, and terms formed with the `#i` reader macro are read as `INDICES` terms when the list following `#i` has three elements instead of two. The following example should make this clear: `5` is the starting index and `3` is the number of elements in the list.

```
#i(A 5 3)
```

is read the same as

```
(INDICES 'A 5 3)
```

which in turn is provably equal to

```
(LIST (INDEX 'A 5)
      (INDEX 'A 6)
      (INDEX 'A 7))
```

Thus, `#i(A 5 3)` is equal to the list of length `3` consisting of index expressions built from `'A`, starting with an "index" of `5` and continuing with successively larger "indices".

The file also has a number of lemmas about the function `INDICES`, for example that the resulting list has no duplicates.

See Section 14.22 for the event file "indices.events".

4.4 HARD-SPECS

This file defines functions for the four-valued logic used through the FM9001 proof (see Chapter 1), as well some basic Boolean and vector functions and some simple functions used in specifications. It also contains basic theorems about these functions.

Among the Boolean functions defined in this file are `B-NOT`, which is just negation, and `B-BUF`, which is the identity except that non-Boolean values are coerced to `T`. Thus, `B-BUF` is actually the same as `BOOLFIX`. Another example is `B-AND3`, which returns the logical AND of three values. Other n-ary Boolean functions (for other arities n and basic Boolean functions) are also defined, as are some basic functions for "compound gates," such as the following:

Definition.

```
(ao2 a b c d)
 =
(b-nor (b-and a b) (b-and c d))
```

The function `BVP` is among the "vector functions" defined here; it returns `T` if and only if its argument is a *bit vector*, i.e., a proper (null-terminated) list of Booleans. There are also "extensions" of the basic Boolean functions to bit vectors. For example, `V-AND` is the bit-wise `AND` of two bit-vectors. Shifting and multiplexing functions are also defined.

Some important functions to be used later in specifications are defined here. For example, the function `V-TO-NAT` converts a bit vector to a corresponding natural number, while the function `NAT-TO-V` goes the other direction by taking a natural number and a given width, producing a corresponding bit vector of that width. The function `V-ZEROP` recognizes those bit vectors whose bits are all `F`. A ripple-carry adder is specified with the recursive definition `V-ADDER`.

Finally, a number of rewrite rules are proved about the functions in this file. For example, the lemma `LENGTH-OF-V-ADDER` says that the length of the `V-ADDER` sum of a carry bit with two bit vectors is one more than the length of the first bit vector. Under "normal" circumstances we would expect the two bit-vector arguments of `V-ADDER` to have the same length, but in fact the definition of `V-ADDER` is such that the length of the result depends solely on the length of the *first* bit-vector input.

See Section 14.23 for the event file "hard-specs.events".

4.5 VALUE

This file defines two very important functions. `VALUE` returns the first value of a key in an association list. For example,

```
(VALUE 'B '((A . 3) (B . 4) (C . 5) (B . 6)))
```

is equal to `4`. The other function is `COLLECT-VALUE`, which collects up the `VALUE`s of a list of keys and returns these values in the corresponding order. For example,

```
(COLLECT-VALUE '(C A B) '((A . 3) (B . 4) (C . 5) (B . 6)))
```

is equal to the list

```
'(5 3 4)
```

The file also contains a number of rewrite rules about these functions. For example, the length of `(COLLECT-VALUE ARGS ALIST)` is proved to be equal to the length of `ARGS`. The utility of some lemmas is less obvious. The following one is perhaps a bit subtle; it instructs the rewriter to attempt to show that the two "alist" arguments are equal when trying to prove the equality of two `COLLECT-VALUE` terms.

Theorem. `EQUAL-COLLECT-VALUE-PROMOTE-ALISTS`

```
(implies
 (equal alist1 alist2)
 (equal (equal (collect-value args alist1)
               (collect-value args alist2))
        t))
```

See Section 14.24 for the event file "value.events".

4.6 MEMORY

The physical implementation of the FM9001 does not include the external memory. However, we do model the memory in the top-level netlist that we reason about. Our memory is modeled using tree structures, with values stored at the tips. In this file we define basic read and write operators for such memories. The memory model presented here is specialized in later files to axiomatize both the register file and the external memory of the FM9001; see Section 4.7 and Section 4.8, respectively.

Memory is modeled in the Nqthm logic by using three shells: the shell constructor `ROM` tags read-only locations of the memory, while the shell constructor `RAM` tags read-write locations and `STUB` represents "unimplemented" portions. The present file contains these shell definitions as well as definitions of functions for reading from and writing to memory and lemmas about these operations. For example, the lemma `NOT-V-IFF-V-ADDR1-V-ADDR2-READ-MEM1-WRITE-MEM1` says that if we perform a write operation before reading the memory, then the write has no effect on the read as long as the write address is different from the read address.

See Section 14.25 for the event file "memory.events".

4.7 DUAL-PORT-RAM

The functions `DUAL-PORT-RAM-VALUE` and `DUAL-PORT-RAM-STATE` define the current outputs and next state of the FM9001 register file. These functions are used in the behavioral specification of the FM9001 register-file primitive in the semantics in our hardware description language. There are comments in the file that describe some of the subtleties in these definitions.

See Section 14.26 for the event file "dual-port-ram.events".

4.8 FM9001-MEMORY

This file contains the specification of the FM9001 external memory. Functions for the outputs and next state of the external memory are defined, analogously to those for the register file in the preceding section, Section 4.7. This behavioral specification is more complicated than the one for the register file because it requires that a specific protocol be used when accessing the memory. For instance, we require that the address lines remain stable during read and write operations.

See Section 14.27 for the event file "fm9001-memory.events".

4.9 TREE-NUMBER

The function `TREE-NUMBER` is used to create indices that are used in the definitions of some of the circuit generator functions. In particular, when we recursively generate circuits, this function is called on to produce module names.

See Section 14.28 for the event file "tree-number.events".

4.10 F-FUNCTIONS

This file can be viewed as a follow-up to the definitions in the file "hard-specs.events" (see Section 4.4), in the sense that it provides more functions and lemmas related to the four-valued logic. Moreover, it contains rules for expanding and reducing calls to these functions in various special cases—e.g., when one or more arguments are known to be Boolean.

The functions defined are mainly the standard logical operators in the underlying four-valued logic. For example, the function `F-AND` is defined to return `T` if its two inputs are both `T`, `F` if either of its two inputs is `F`, and the 'unknown' value `(X)` otherwise. In general, these functions "fix" their values to one of `T`, `F`, or `(X)`. However, there are two special functions `FT-BUF` and `FT-WIRE` that may return the floating or `(Z)` value. These are intended for modeling tri-state logic.

The following functions are defined: `F-BUF`, `F-AND`, `F-AND3`, `F-AND4`, `F-NOT`, `F-NAND`, `F-NAND3`, `F-NAND4`, `F-NAND5`, `F-NAND6`, `F-NAND8`, `F-OR`, `F-OR3`, `F-OR4`, `F-NOR`, `F-NOR3`, `F-NOR4`, `F-NOR5`, `F-NOR6`, `F-NOR8`, `F-XOR`, `F-XOR3`, `F-EQUV`, `F-EQUV3`, `F-IF`, `FT-BUF`, `FT-WIRE`, `F-PULLUP`. These functions are generally used to define the semantics of the primitive operators of our hardware description language.

In addition, this file contains a number of lemmas related to these functions. For example, the lemma `EXPAND-F-FUNCTIONS` (which is usually disabled) asserts the equality of each f-function call to its body. This permits the call to be opened up by the Nqthm rewriter when proving low-level facts about the four-valued logical operators. Also included are various special rewrite rules encoding various standard Boolean reductions; an example is `F-AND-REWRITE` rewriting `(F-AND F X)` to `F`.

One function given special attention in this file is `F-BUF`. It is used to define the semantics of the buffer primitives. It is a unary function that "fixes" its input to one of `T`, `F`, or `(X)`. For Boolean inputs, it is a no-op. There are various rewrite rules encoding properties of `F-BUF`.

The four-valued functions all reduce to standard Boolean operators when their arguments are Boolean. This is captured in a collection of lemmas, such as `F-GATES=B-GATES`. These provide rewrite rules that replace four-valued logic functions such as `F-NOT` with the corresponding Boolean logic functions such as `B-NOT` when the inputs are known to be Boolean.

Finally, each of the four-valued functions has a corresponding bit-vector version. These functions are defined—mapping the bit function over the input lists—and various helpful lemmas are proved.

See Section 14.29 for the event file "f-functions.events".

Chapter 5

THE SYNTAX AND SEMANTICS OF OUR HARDWARE DESCRIPTION LANGUAGE

Here we introduce the function DUAL-EVAL, which gives an operational (simulator) style semantics for our hardware description language, together with recognizers for well-formed netlists in this language. This chapter concludes with the function LISP-NETLIST that makes our mechanically-generated netlist descriptions more readable.

5.1 DUAL-EVAL

A module in our HDL is a representation of a hardware circuit design. Conceptually, the *meaning* of the module is the computation performed by that circuit for given inputs. This is formalized operationally by a collection of functions that *simulate* circuit modules for specific values of the input variables, and produce the appropriate values for the outputs and state. These interpreter functions give meaningful values only for well-formed circuit descriptions. Our syntactic constraints assure that a well-formed circuit module body has its occurrences ordered such that the outputs can be computed efficiently.

It is useful to imagine taking snapshots of a given circuit right around the rising edge of each clock cycle. In this model, the values on the circuit's wires will have been stable for at least a little while before that edge, including the primary inputs. In particular, the output wires have achieved values during the clock cycle and can be "photographed" just before the clock ticks. The state-holding devices—latches, register file, and memory—are updated very shortly after the rising edge, based on the values that their inputs stabilized to over the previous clock cycle; we imagine "photographing" their internal states just after the clock's rising edge. Thus, we define our semantics by a function that takes as arguments (1) the values attained during a given clock cycle on the primary input wires and (2) the internal state latched in at the beginning of that clock cycle, and returns the next "snapshots": the values on the output wires just before the end of the cycle, and the internal state immediately after the next rising edge. This function, called `DUAL-EVAL`, computes two results: outputs and next state. `DUAL-EVAL` is so-called because it uses two passes over the circuit netlist to compute these values, one pass to update the internal wire values, and another pass to update the internal state. We explain `DUAL-EVAL` in some detail below.

Now imagine applying this `DUAL-EVAL` function repeatedly, starting with a given internal state and taking a list of input vectors to be read in during each clock cycle. The result of this repeated application is a list of "snapshots," that is, a list of pairs each containing the output vector and internal state at successive rising edges of the clock, in the sense described above. This iteration of `DUAL-EVAL` is performed by the function `SIMULATE`. In conventional jargon, the `DUAL-EVAL` HDL allows Mealy machines to be specified. `SIMULATE` is the corresponding simulator, that is, the corresponding operational semantics for our HDL.

Let us turn now to the "snapshot" function `DUAL-EVAL` described above. A call to `DUAL-EVAL` takes the following form:

```
(DUAL-EVAL flag function arguments state netlist)
```

A flag value of `0` or `2` determines, respectively, whether the interpreter computes the values of the output lines or the new values of the state-holding components. Values of `flag` of `1` and `3` are used to signal recursive evaluation of occurrence lists. The other arguments are as follows when `flag` is `0` or `2`:

- `function`: the name of the module to be evaluated;
- `arguments`: the list of values of the input lines for this module evaluation;
- `state`: a list of current values of the state-holding components of the module; and
- `netlist`: the netlist in which module `function` is defined.

For example, assume that we have the following netlist `NETLIST`, representing a *full-adder* module in the HDL.

```
'((FULL-ADDER (A B C)
              (SUM CARRY)
              ((T0 (SUM1 CARRY1) HALF-ADDER (A B))
               (T1 (SUM CARRY2)  HALF-ADDER (SUM1 C))
               (T2 (CARRY)       B-OR (CARRY1 CARRY2)))
              NIL)
  (HALF-ADDER (A B)
              (SUM CARRY)
              ((G0 (SUM)   B-XOR (A B))
               (G1 (CARRY) B-AND (A B)))
              NIL))
```

Regardless of the value of `STATE`, the following equality holds:

```
(DUAL-EVAL 0 'FULL-ADDER (LIST T F T) STATE NETLIST) = (LIST F T)
```

That is, `DUAL-EVAL` here returns the list of values of the `FULL-ADDER` outputs `SUM` and `CARRY`, when the inputs `A`, `B`, and `C` take values `T`, `F`, and `T`, respectively. The preceding equality is independent of the value of `STATE` because there are no state-holding elements in `NETLIST` in this case.

The call

```
(DUAL-EVAL 2 'FULL-ADDER (LIST T F T) STATE NETLIST)
```

returns the list of updated values of the *state-holding components* of module `FULL--ADDER` on the same arguments, and in this case returns `NIL` since the `FULL-ADDER` module has no state-holding components.

The examples above have all assumed Boolean values. However, the underlying evaluation model used by `DUAL-EVAL` is four-valued, with logical values of `T`, `F`, `(Z)`, and `(X)`; `(Z)` is the "floating value" and `(X)` is "undefined." All of the primitives (and, hereditarily, all defined modules) are defined on the four logical values.

The values produced by `DUAL-EVAL` are computed by a recursive evaluation of a module definition and its component modules. Calls to HDL primitives are handled by the subsidiary functions `DUAL-APPLY-VALUE` and `DUAL-APPLY-STATE`, which look up the semantics of the primitive in a database (see Section 2.9) and apply that semantics to the arguments.

The syntactic constraints on the language assure that this can be done efficiently, in two passes through the occurrence list. Since all of our functions, including `DUAL-EVAL`, are defined within the Boyer-Moore logic, we can use the logic's interpreter to simulate them for explicit input values. We use this capability extensively to "debug" our circuit designs before investing the time to prove the correctness of our circuit designs.

The treatment of time in our interpreter model is simplistic. Each application of `DUAL-EVAL` simulates one (implicit) "tick" of the global clock. Restrictions on the language, such as the prohibition of combinational loops, guarantee that the evaluation will reach a "stable state" within one clock tick. We also define the "multi-clock" function named SIMULATE.

Definition.
```
(simulate fn inputs state netlist)
 =
(if (nlistp inputs)
    nil
  (let ((value (dual-eval 0 fn (car inputs) state netlist))
        (new-state (dual-eval 2 fn (car inputs) state netlist)))
    (cons (list value new-state)
          (simulate fn (cdr inputs) new-state netlist))))
```

This function models the recursive application of DUAL-EVAL over time by returning a list of 2-element lists of the form (LIST outputs state), where outputs and state are the results of calling DUAL-EVAL with flags 0 and 2, respectively. The argument INPUTS is a *list* of input vectors, one for each clock tick. The function SIMULATE-DUAL-EVAL-2 is similar, except that it returns a single (final) state rather than a list of output-state pairs.

In addition to the definitions of DUAL-EVAL and SIMULATE, this file also contains some rewrite rules useful for opening up the definition of DUAL-EVAL in particular circumstances.

See Section 14.30 for the event file "dual-eval.events".

5.2 PREDICATE-HELP

The functions in this file are used in the files "predicate-simple.events" and "predicate.events".

See Section 14.31 for the event file "predicate-help.events".

5.3 PREDICATE-SIMPLE

This file contains a recognizer for well-formed netlists. The recognizers in this file are defined to be as simple as possible, and thus return Boolean results.

The predicate TOP-LEVEL-PREDICATE-SIMPLE is a simpler version of the circuit predicate TOP-LEVEL-PREDICATE in the file "predicate.events". These predicates only return a symbol indicating what main property was violated. The circuit recognizer predicates in file "predicate.events" return more useful information, but the functions there are much more complex.

Definition.

```
(top-level-predicate-simple netlist)
 =
(if (not (netlist-syntax-simple-okp netlist))
    'netlist-syntax-simple-okp-error
  (if (not (simple-dependency-table netlist))
      'dependency-table-simple-error
    (if (not (netlist-type-check-simple-okp netlist))
        'netlist-type-check-simple-okp-error
      (if (not (netlist-state-types-simple netlist))
          'netlist-state-types-simple-error
        (if (not (netlist-loadings-and-drives-simple netlist))
            'netlist-loadings-and-drives-simple-error
          t)))))
```

The ordering of the tests in the definition of TOP-LEVEL-PREDICATE-SIMPLE is important. That is, before we check for type errors, we insist that the netlist syntax is correct. This predicate is not sufficiently general to recognize circuits with tri-state components—it does not permit a wire to be used in a bi-directional way.

We do not use these circuit predicates during our proofs; however, once we have a netlist we check its well-formedness using these predicates.

Note: we know that the two predicates TOP-LEVEL-PREDICATE-SIMPLE and TOP-LEVEL-PREDICATE (discussed in the next section) do not recognize exactly the same language. Thus we use both predicates when we test a module for well formedness, except in the case of modules that include tri-state circuits when only TOP-LEVEL-PREDICATE is used.

See Section 14.32 for the event file "predicate-simple.events".

5.4 PREDICATE

Here we define a predicate that recognizes well-formed circuits, formulated in the Hardware Description Language (HDL). The predicate TOP-LEVEL-PREDICATE has one parameter, a netlist (the normal HDL circuit description). The value of TOP-LEVEL-PREDICATE is true (T) for a well-formed netlist, but is a description of all errors detected for an ill-formed netlist. Among the errors detected are excessive fan-out, wire type mismatches, clock distribution, arity problems, duplicate names, inappropriate names, missing fields, and combinational loops. An itemized list appears below.

A second predicate LSI-TOP-LEVEL-PREDICATE has additional well-formedness checks for LSI Logic's requirements. This is important because we sometimes translate our HDL into LSI Logic's language, NDL. This predicate has three parameters:

`netlist:` a netlist, in which all names are litatoms

`token-size:` maximum allowed name length (should be 64)

`max-hierarchical-name-length:`
maximum allowed length for LSI Logic's hierarchical names (should be 255)

First let us look at two small examples. At the end of this description we present a much larger example. Although the output of `TOP-LEVEL-PREDICATE` is intended to be self-explanatory, we annotate both input and output below with semicolons to indicate comments in the usual Lisp style.

The first example illustrates syntax errors.

```
(top-level-predicate
  '((FULL-ADDER (C A B)
                (SUM CARRY)
                ((G0 (SUM1 CARRY1) HALF-ADDER (A B))
                 (G1 (SUM CARRY2) HALF-ADDER (SUM1 C EXTRA)) ; too many inputs
                 (G2 (CARRY) B-OR (CARRY1 CARRY2)))
                NIL)
    (HALF-ADDER (A B)
                (SUM CARRY)
                ((37 (SUM) B-XOR (A B)) ; 37 is not a valid occurrence name
                 (G1 (CARRY) B-AND (A B)))
                NIL)))

 =

;; There are two syntax errors, one in each module.
 (NET-ERROR
  'NETLIST-SYNTAX-ERRORS
  (LIST
   ;; first, the extra-input error in module FULL-ADDER
   (NET-ERROR
    '(MODULE FULL-ADDER)
    (LIST
     ;; the first error is in the occurrences field of FULL-ADDER
     (NET-ERROR 'MODULE-OCCURRENCES
      (LIST
       ;; first note that occurrence G1 has the wrong number of inputs
       (NET-ERROR
        '(OCCURRENCE G1)
        (LIST
             (NET-ERROR 'OCCURRENCE-FUNCTION
                        (LIST (NET-ERROR 'WRONG-NUMBER-OF-INPUTS
                                         '(OCC-FUNCTION HALF-ADDER
                                                        (EXPECTED 2)
                                                        (GOT 3)))))))
       ;; next note that EXTRA is used but was never defined
```

```
      (NET-ERROR 'NON-DEPENDS-NOT-IN-SIGNALS
                 '(EXTRA))))))
   ;; now, for the error in module HALF-ADDER
   (NET-ERROR
    '(MODULE HALF-ADDER)
    (LIST
     ;; the error is in the occurrences field here too
     (NET-ERROR 'MODULE-OCCURRENCES
      (LIST
       ;; 37 is a bad occurrence name
       (NET-ERROR '(OCCURRENCE 37)
                  (LIST (NET-ERROR 'OCCURRENCE-NAME
                                   (NET-ERROR 'BAD-NAME 37))))))))))
```

Here is a second example. This one illustrates an error that is more "semantic" in nature.

```
(top-level-predicate
  '((FOO (A B)
         (C)
         ((G1 (D) T-BUF (A B)) ;; so, D is a tri-state wire
          (G2 (C) B-NOT (D)))  ;; so, D is a Boolean wire
         NIL)))

 =

;; The label NETLIST-ERRORS indicates a ``semantic'' error.
;; In this case, D is a tri-state output wire for the T-BUF
;; but it is a Boolean input to the B-NOT.
 (NET-ERROR
  'NETLIST-ERRORS
  (LIST
   (NET-ERROR
    '(MODULE FOO)
    (LIST
        (NET-ERROR 'IO-TYPE-CONFLICTS
                   (LIST (NET-ERROR '(SIGNAL D)
                                    '((INPUT-TYPE BOOLP)
                                      (OUTPUT-TYPE TRI-STATE))))))))
```

Let us turn now to details about the predicates. The netlist predicates work in two stages. First, they (functions NETLIST-SYNTAX-OK and LSI-NETLIST--SYNTAX-OK) check the netlist for syntax errors. Then, they check for other errors by attempting to construct a database of netlist properties (function NETLIST--PROPERTIES), similar to the primitive database (see Section 2.9). This database may be viewed as an extension of the constant (PRIMP-DATABASE) (defined in Section 5.1) to a table for all the modules in the netlist, though the properties of interest need to be given as a parameter to NETLIST-PROPERTIES. That is, this function builds a table associating with each module name various properties such as drives, input names, loads, and so on.

Also included is function NETLIST-DATABASE, which also constructs a database of netlist properties but does not take a list of desired properties as an input. Instead, the netlist database returned by this function contains all the properties appearing in PRIMP-DATABASE, except for LSI-NAME, NEW-STATES, and RESULTS properties. NETLIST-DATABASE assumes there are no syntax errors in the netlist.

In addition, the following functions return single properties of a netlist, module, or primitive.

Number of gates, pads, primitives, or transistors:	
	`primitive-count`

Output dependencies:	`dependency-table`
	`output-dependencies`
	`out-depends`

Signal types:	`netlist-type-table` `I/O-types`
State types:	`netlist-state-types` `state-type-requirement`
Loadings and drives:	`netlist-loadings-and-drives` `loadings-and-drives`

The functions above were defined under the implicit assumption that the primitive database is well-formed, as well as the netlist we are analyzing. We check the well-formedness of the database with function PRIMP-DATABASE-PREDICATE, as we now describe.

5.4.1 The Primitive Database

The primitive database is checked by the function PRIMP-DATABASE-PREDICATE for the presence and consistency of *properties*, that is, values of the fields associated with each primitive. The primitive properties and some requirements on them are:

DELAYS. Signal delay as a function of loading. There should be an entry for each output. Each entry has the following form.

```
  ((<low to high slope> <low to high intercept>)
   (<high to low slope> <high to low intercept>))
| <input name>
| (OR <input name>*)
```

An input name means that the output has the same delay as the input. The `OR` form means that the output has the same delay as one of the named inputs.

DRIVES. Measure of the loading a signal can tolerate. There should be an entry for each output. Each entry has the following form, where the optional designator "mA" stands for milliAmps.

```
  <number>
| (<number> mA)
| <input name>
| (MIN <input name>*)
```

An input name means that the output has the same drive value as the input. The `MIN` form means that the output has the least drive value of the named inputs.

INPUT-TYPES. There should be one type for each input. Acceptable types are `boolp`, `clk`, `free`, `level`, `parametric`, `tri-state`, `ttl`, `ttl-tri-state`. Only `tri-state` is acceptable for primitive `t-wire`. Bi-directional I/O signals should be of type `tri-state` or `ttl-tri-state`.

INPUTS. List of input signal names.

LOADINGS. There should be an entry for each input. Each entry has the following form, where the optional designator "pf" stands for picoFarads.

```
<number> | (<number> pf)
```

LSI-NAME. LSI Logic's name for the primitive, possibly with reordered inputs. Reordered input signals should include all and only the signals in the `INPUTS` property.

NEW-STATES. The property, when evaluated by `DUAL-EVAL`, should give a single state value or a list of two or more state values, corresponding to the `STATES` property. Each new state value should depend only on `STATES` and `INPUTS`. If this property is present, `STATES` and `STATE-TYPES` should also be present.

OUT-DEPENDS. There should be an entry for each output. Each entry should be the subset of `INPUTS`, on which the corresponding output depends. An output may not depend on other signals if it is a constant or if it is extracted from the state, representing storage devices.

OUTPUT-TYPES. There should be one type for each output. Acceptable types are `boolp`, `clk`, `level`, `parametric`, `tri-state`, `ttl`, `ttl-tri-state`. The type given means that the output has the same type as the input. Only `tri-state` is acceptable for primitive `t-wire`. Bi-directional I/O signals should have the same type as they do in `INPUT-TYPES`, either `tri-state` or `ttl-tri-state`.

OUTPUTS. List of output signal names.

RESULTS. The property, when evaluated by `DUAL-EVAL`, should give one value for each output. Each output value should depend only on `STATES` and the `OUT-DEPENDS` entry for the output.

STATE-TYPES. The value should be a single state type or a list of two or more state types, corresponding to the `STATES` property. Acceptable state types are composed from the following; an exception is that there is a numeric (controller) component to the state of the `mem-32x32` primitive.

```
  (RAM (boolp*))
| (ROM (boolp*))
| (STUB (boolp*))
| boolp
```

If this property is present, `STATES` and `NEW-STATES` should also be present.

STATES. The value should be a single name, or a list of two or more names. A name represents a storage device. If this property is present, `NEW-STATES` and `STATE-TYPES` should also be present.

GATES. Number of gates in the primitive.

PADS. Number of pads in the primitive. This property is not required.

PRIMITIVES. Number of primitives (usually one). The value is zero for pseudo-primitives id and t-wire.

TRANSISTORS. Number of transistors in the primitive.

5.4.2 Syntax Requirements

The syntax of a netlist in the `DUAL-EVAL` HDL is given in Figure 5.1. The syntax is described mostly in an informal Backus-Naur Form (BNF). To be sure that a netlist is meets the syntactic requirements, the function `TOP-LEVEL-PREDICATE` can be used to check a concrete netlist using the Nqthm evaluator `R-LOOP`.

The notation is as follows:

Figure 5.1: HDL Syntax

```
<netlist>               ::=   ( <module>* )

<module>                ::=   ( <module name> <module inputs>
                                              <module outputs>
                                              <module occurrences>
                                              <module state names>
                                              [ <annotation> ] )

<module name>           ::=   <name>

<module inputs>         ::=   ( <name>* )

<module outputs>        ::=   ( <name>* )

<module occurrences>    ::=   ( <occurrence>+ )

<module state names>    ::=   NIL | <name> | ( <name> <name>+ )

<annotation>            ::=   ( ( <key> . <value> )* )

<occurrence>            ::=   ( <occurrence name> <occurrence outputs>
                                                  <occurrence function>
                                                  <occurrence inputs>
                                                  [ <annotation> ] )

<occurrence name>       ::=   <name>

<occurrence outputs>    ::=   ( <name>* )

<occurrence function>   ::=   <name>

<occurrence inputs>     ::=   ( <name>* )

<name>                  ::=   <simple name> | <indexed name>

<simple name>           is    a litatom.

<indexed name>          is    a shell object, constructed by
                              (index <simple name> <number>),
                              where <number> is a numberp.

<key>                   is    unrestricted.

<value>                 is    unrestricted.
```

- <x> ::= y means that <x> is a nonterminal symbol and its definition is y. Grammar symbols, nonterminal and terminal, are surrounded by angle brackets <>.
- Within the definition y:
  - The vertical bar | means "or" and separates alternative definitions.
  - Text in square brackets [...] is optional.
  - A * following a grammar symbol or a sequence of symbols in parentheses indicates that the symbol or sequence can appear zero or more times.
  - A + following a grammar symbol or a sequence of symbols in parentheses indicates that the symbol or sequence can appear one or more times.

Additional syntax requirements are:

- No <module name> is the same as a primitive name, and there are no duplicate <module name>s in the <netlist>.
- There are no duplicate signal names within a <module> except for I/O-signals (intersection of <module inputs> and <module outputs>). The signals of a module are its inputs and <occurrence outputs> from all its occurrences.
- Within a <module>, all <module outputs> are generated once and only once as <occurrence outputs>.
- All <occurrence inputs> within a <module> are signals of the module.
- <module state names> are a subset of the module's <occurrence name>'s.
- There are no duplicate <occurrence name>'s within a module.
- Every <occurrence function> names either a primitive or a <module> defined in the netlist after the <module> where the <occurrence function> appears.
- Numbers of <occurrence inputs> and <occurrence outputs> are the same as the numbers of inputs and outputs required by the definition of the module or the description of the primitive named by the <occurrence function>.

LSI Logic's Network Description Language (NDL) has the following additional syntax requirements:

- A <name> is a standard litatom beginning with an upper case letter and containing only upper case letters, digits, hyphens (-), and underscores (_). A <name> can contain at most 64 characters (token-size). No name can be an NDL keyword.

- LSI Logic uses hierarchical names to provide a unique name for each signal in a circuit. The name is composed by appending `/<occurrence name>` for each occurrence in an instance tree until a primitive is reached; at that point `/<name>` is appended, where `<name>` is the name of a signal in the occurrence that is an instance of the primitive. No hierarchical name can contain more than 255 characters (`max-hierarchical-name-length`).

- No signal name (module input or occurrence output, including module outputs) can be the same as an occurrence name except that if an occurrence has only one output, the output name can be the same as the name of that occurrence.

- No signal name or occurrence name in a module can be the same as the name of any function referenced in that module. If the function is a primitive, LSI Logic's name for it is the one that cannot be used as a signal or occurrence name.

5.4.3 Other Well-Formedness Requirements

Signal Types. Type requirements for all signals in a netlist are inferred from the types of its primitives. Contradictory type requirements give an error. I/O signals should have `tri-state` output types and either `tri-state` or free input types. All modules with `T-WIRE` type inputs should have `tri-state` input types. Otherwise, the output type of a signal is required to be a subtype of its input type.

Loadings and Drives. When a signal is generated, it has a limited strength, or drive. When the signal is used, the drive is depleted by some amount, the loading. In a well-formed netlist, the loading on any signal cannot exceed its drive. An overloaded signal gives an error. This computation provides a crude measure of acceptable fanout.

Output Dependencies. For each signal in the netlist, we compute an output dependency list. An output dependency is the name of an input on which the signal combinationally depends. Output signals do not depend on the inputs when they are constants or are derived solely from the internal state of a module.

It is an error if a signal `S` (occurrence input or module output) depends on an unknown signal. In particular, every occurrence input must be "already defined" in the sense that it either appears among the module's inputs or that it appears in a preceding occurrence's output list. The output dependency computation gives errors for circularities (combinational wiring loops or feedback) in circuit descriptions. This restriction demands that the value of every `<occurrence output>` can be computed in a single pass.

State Types. The state or list of states, appearing in a module definition, should correctly reflect the state requirement of the module. This is checked by constructing a mapping from module occurrence names to state types. For each occurrence of a submodule or primitive with state, the name of the occurrence is mapped to the state type required by the submodule or primitive. The state type of a primitive is given in the primitive database. It is an error if the states in the module definition do not include all and only the occurrence names in the mapping. It is an error if the states listed in a module definition are not the same as the states for which types are computed, ultimately from primitive state types.

Tri-State Signal Checks. A signal that is an input to a `t-wire` primitive cannot be fanned out. In addition, a signal can be an input to at most one `t-wire` primitive. Finally, no I/O signal (i.e., signal that is in both the input and output lists of the module) can be an input or an output of the primitive `ID` (which declares aliases for signal names).

See Section 14.33 for the event file "predicate.events".

5.5 PREDICATE-TESTS

This file contains many netlists with a variety of problems. We use these netlists to check that our netlist predicate, `TOP-LEVEL-PREDICATE`, recognizes these netlists as ill-formed. This file in not actually part of the FM9001 run.

See Section 14.34 for the test file "predicate.tests".

5.6 PRIMITIVES

Perhaps the most important command in this file is `(RESULT-AND-STATE-LEMMAS)`. It generates a large number of rewrite rules about the primitive (built-in) `DUAL-EVAL` hardware modules. Note that these lemmas are all generated automatically by the command above, which actually calls on several macros that are defined in the file "primitives.lisp" (Section 2.10). The few additional events in this file are mainly technical in nature, so we will not discuss them here. Instead, let us take a look at the events generated by `(RESULT-AND-STATE-LEMMAS)` for a flip-flop primitive, `FD1S`. In each case the event is preceded by a short explanatory comment.

Our convention for naming the predicate that recognizes valid subnets in a netlist is to concatenate the "`&`" character to the end of the module name. Note that every netlist is "valid" for primitives, because their definition is built into the semantics (`DUAL-EVAL`, see Section 5.1) of our hardware description language. In particular, every netlist correctly "represents" the primitive `FD1S`.

```
(DEFN FD1S& (NETLIST) T)
```

We disable the definition of each netlist recognizer.

```
(DISABLE FD1S&)
```

The minimal (sub)netlist generated for correctly defining `FD1S` is empty, since `FD1S` is a primitive.

```
(DEFN FD1S$NETLIST NIL NIL)
```

The following `$VALUE` lemma gives the semantics for the output wires of an `FD1S` primitive in terms of a simple specification function. That is, it says that the values on the two output wires of this device are (in order) the value of the state (suitably "buffered") followed by the negated value of the state.

```
(PROVE-LEMMA FD1S$VALUE (REWRITE)
    (IMPLIES (FD1S& NETLIST)
             (EQUAL (DUAL-EVAL 0 'FD1S (LIST D CP TI TE) STATE
                          NETLIST)
                    (CONS (F-BUF STATE) (CONS (F-NOT STATE) 'NIL))))
    ((ENABLE FD1S& PRIMP DUAL-APPLY-VALUE)
     (EXPAND (DUAL-EVAL 0 'FD1S (LIST D CP TI TE) STATE NETLIST))))
```

We disable all `$VALUE` lemmas.

```
(DISABLE FD1S$VALUE)
```

The following `$STATE` lemma asserts that the next internal state of the `FD1S` device is the value of the test-in input `TI` if the test-enable line `TE` is high; otherwise, it is the value of the data input `D`.

```
(PROVE-LEMMA FD1S$STATE (REWRITE)
    (IMPLIES (FD1S& NETLIST)
             (EQUAL (DUAL-EVAL 2 'FD1S (LIST D CP TI TE) STATE
                          NETLIST)
                    (F-IF TE TI D)))
    ((ENABLE FD1S& PRIMP DUAL-APPLY-STATE)
     (EXPAND (DUAL-EVAL 2 'FD1S (LIST D CP TI TE) STATE NETLIST))))
```

The `$STATE` lemmas are kept disabled.

```
(DISABLE FD1S$STATE)
```

See Section 14.35 for the event file "primitives.events".

5.7 UNBOUND

This file defines functions UNBOUND-IN-BODY and ALL-UNBOUND-IN-BODY that check that a given name or (respectively) names do not occur as an output of any occurrence in a given "body" (occurrence list). The final two lemmas in the file formalize the idea that if a name is not an output of an occurrence list, then it is not bound by the evaluation of the module using DUAL-EVAL.

When reasoning by induction about recursive module generators, it is often necessary to know that one or more signal names in the body (occurrence list) being generated do not appear as outputs, as bound by DUAL-EVAL. This knowledge, together with the lemmas referred to above, allows us to deduce that the values of those signal names in a binding list produced by running DUAL-EVAL is the same as their values in the input binding list.

See Section 14.36 for the event file "unbound.events".

5.8 VECTOR-MODULE

This file defines an induction scheme for later use in proofs about certain "vector modules," along with vector versions of the primitives B-BUF, B-OR, B-XOR, PULLUP, and T-WIRE. These are all defined using the macro VECTOR-MODULE; see Section 2.8 for an example of what this really does.

See Section 14.37 for the event file "vector-module.events".

5.9 TRANSLATE

The function LISP-NETLIST converts an Nqthm netlist constant, which may include INDEX shells, to an Nqthm constant that does not contain such shell objects. The following example is obtained by mere evaluation, using the Nqthm logic "read-eval-print" loop, R-LOOP:

```
(lisp-netlist
 (list (index 'v-buf 2)
       (list (index 'a 0) (index 'a 1))
       (list (index 'y 0) (index 'y 1))
       (list (list (index 'g 0)
                   (list (index 'y 0))
                   'b-buf
                   (list (index 'a 0)))
             (list (index 'g 1)
                   (list (index 'y 1))
                   'b-buf
                   (list (index 'a 1))))
```

```
        nil))

  =

 '(V-BUF_2 (A_0 A_1)
           (Y_0 Y_1)
           ((G_0 (Y_0) B-BUF (A_0))
            (G_1 (Y_1) B-BUF (A_1)))
           NIL)
```

Finally, we discuss the function COLLECT-PRIMITIVES. Given a module name and a netlist that defines a module with that name, one can imagine the following graph. Its nodes are labeled with module names and its root is the given module name. The list of children of any node is simply the list of all module names referenced in the given module's occurrence list. Notice that the tips of this graph are all primitives, at least for any well-formed netlist. The function COLLECT-PRIMITIVES takes as inputs a module name and a netlist, and returns the list of all primitive module names that are leaves of the graph described above.

See Section 14.38 for the event file "translate.events".

Chapter 6

TWO SIMPLE EXAMPLES

The following two sections are not used elsewhere. Rather, they provide simple examples of the use of our HDL.

6.1 EXAMPLES

In this file we show how to verify two simple combinational modules, which can be used to build an adder. Also we develop a one-bit tri-state data bus, including specification and proof of correctness.

See Section 14.39 for the event file "examples.events".

6.2 EXAMPLE-V-ADD

This file contains an exercise in developing an n-bit ripple-carry adder, including specification and proof of correctness, from `HALF-ADDER` and `FULL-ADDER` modules defined in the previous file.

See Section 14.40 for the event file "example-v-add.events".

Chapter 7

MISCELLANEOUS COMBINATIONAL HARDWARE MODULES AND N-BIT REGISTERS

This chapter describes a number of miscellaneous hardware modules. We develop the theory for the propagate-generate ALU and describe how we generate simple registers.

7.1 PG-THEORY

These events develop a propagate-generate specification function `TV-ADDER` that is later implemented with hardware modules. We prove that `TV-ADDER` is equal to a ripple-carry adder `V-SUM`. Ripple-carry adder implementations take time linear in the size of their inputs, but a propagate-generate adder takes logarithmic time.

Here is a crash course on the basic algorithm for those not so familiar with adder design. The idea of a propagate-generate adder is to separate out the following two notions, *propagate* and *generate*. The *generate* from the addition of two bit vectors is `T` if and only if the addition of the two bit vectors produces an output carry regardless of the input carry value. The *propagate* bit is similar, except that when this bit is set, we only know that there will be a carry out under the assumption that the carry in is set. An important observation is that the propagate and generate may be produced in logarithmic time by a divide-and-conquer algorithm. Since the carry out may be computed rapidly (in constant time) from the propagate and generate and the carry in, it is not hard to see that the sum operation may be carried out in log time.

The function TV-ADDER is crucial here. Its definition assumes that the TREE--SIZE of the TREE argument is expected to be equal to the length of both inputs A and B. It returns a list whose first two bits are respectively the propagate and generate from the addition of A and B, and whose remaining elements are the sum.

See Section 14.41 for the event file "pg-theory.events".

7.2 TV-IF

This file defines a parameterized family of multiplexors whose control value is buffered when necessary as it is fanned out, using a tree to specify the shape of the control fan-out. The specification function for this multiplexor (family) is the four-valued vector "if" from "f-functions" (see also Section 4.10). The final theorem in this file states the correctness of these multiplexors.

Theorem. TV-IF$VALUE

```
(implies
 (and (tv-if& netlist tree)
      (properp a) (properp b)
      (equal (length a) (tree-size tree))
      (equal (length b) (tree-size tree)))
 (equal (dual-eval 0
                   (index 'tv-if (tree-number tree))
                   (cons c (append a b))
                   state
                   netlist)
        (fv-if c a b)))
```

Note that we do not use our vector macros in this file, because they were not developed for these tree-based module generators.

See Section 14.42 for the event file "tv-if.events".

7.3 T-OR-NOR

The goal of the following events is to define a fast zero detector. A simple zero detector may be built simply by OR-ing together all bits of a bit vector and then negating the result. However, we want to use negated logic where possible, for performance reasons: NOR gates are typically faster than OR gates. Moreover, we want the propagation delay to be proportional to the log of the length of the input vector, and hence we wish to use a tree-based algorithm such as is used in the preceding section, Section 7.2.

At the top level, then, we may view the fast zero detector as a multi-input NOR module. Our requirements for building such a module are that it be built out of negated logic (NAND and NOR gates) as well as that it have logarithmic propagation delay. In order to satisfy these requirements, we use an algorithm that generates either OR or NOR modules, according to a parity argument. When we finally use this algorithm to build a fast zero detector, we choose an appropriate parity based on the height of the tree of gates.

Here are two samples of the fast zero detector, for eight bits and for three bits respectively. Recall from Section 5.9 that the function `LISP-NETLIST` removes calls of the shell constructor `INDEX` in favor of underscore characters.

```
(lisp-netlist (tv-zerop$netlist (make-tree 8)))

=

'((TV-ZEROP_8 (IN_0 IN_1 IN_2 IN_3 IN_4 IN_5 IN_6 IN_7)
              (OUT)
              ((GO (OUT) T-NOR_8 (IN_0 IN_1 IN_2 IN_3 IN_4 IN_5 IN_6 IN_7)))
              NIL)
  (T-NOR_8 (A_0 A_1 A_2 A_3 A_4 A_5 A_6 A_7)
           (OUT)
           ((LEFT (LEFT-OUT) T-OR_4 (A_0 A_1 A_2 A_3))
            (RIGHT (RIGHT-OUT) T-OR_4 (A_4 A_5 A_6 A_7))
            (OUTPUT (OUT) B-NOR (LEFT-OUT RIGHT-OUT)))
           NIL)
  (T-OR_4 (A_0 A_1 A_2 A_3)
          (OUT)
          ((LEFT (LEFT-OUT) T-NOR_2 (A_0 A_1))
           (RIGHT (RIGHT-OUT) T-NOR_2 (A_2 A_3))
           (OUTPUT (OUT) B-NAND (LEFT-OUT RIGHT-OUT)))
          NIL)
  (T-NOR_2 (A_0 A_1)
           (OUT)
           ((LEAF (OUT) B-NOR (A_0 A_1)))
           NIL)
  (B-BUF (IN)
         (OUT)
         ((GO (OUT- OUT) B-NBUF (IN)))
         NIL))
```

```
(lisp-netlist (tv-zerop$netlist (make-tree 3)))

=

'((TV-ZEROP_14 (IN_0 IN_1 IN_2)
               (OUT)
               ((G0 (OUT-) T-OR_14 (IN_0 IN_1 IN_2))
                (G1 (OUT) B-NOT (OUT-)))
               NIL)
  (T-OR_14 (A_0 A_1 A_2)
           (OUT)
           ((LEFT (LEFT-OUT) T-NOR_1 (A_0))
            (RIGHT (RIGHT-OUT) T-NOR_2 (A_1 A_2))
            (OUTPUT (OUT) B-NAND (LEFT-OUT RIGHT-OUT)))
           NIL)
  (T-NOR_1 (A_0)
           (OUT)
           ((LEAF (OUT) B-NOT (A_0)))
           NIL)
  (T-NOR_2 (A_0 A_1)
           (OUT)
           ((LEAF (OUT) B-NOR (A_0 A_1)))
           NIL)
  (B-BUF (IN)
         (OUT)
         ((G0 (OUT- OUT) B-NBUF (IN)))
         NIL))
```

See Section 14.43 for the event file "t-or-nor.events".

7.4 FAST-ZERO

Although the propagate-generate adder described in Section 7.1 provides its outputs quickly (in time logarithmic in the input width), nevertheless some of those outputs have greater delay than others. For instance, the least significant bit of the sum is produced very rapidly, while the most significant bit takes longer. One of the critical paths in the ALU, of which the present zero-detector is a part, is a path containing the most significant bit output from the ALU. Therefore, we bias the operation of the zero-detector so that the delays from the two most significant input bits are minimized. Interestingly, further analysis using LSI Logic's timing analysis tool showed that no gain was realized using this approach, because the tree of gates in the zero detector gained an additional gate delay on the third most significant output bit.

See Section 14.44 for the event file "fast-zero.events".

7.5 V-EQUAL

Our word-equality tester simply lays down XOR gates for bitwise comparison, and then uses the zero detector from Section 7.3 to finish the test.

See Section 14.45 for the event file "v-equal.events".

7.6 V-INC4

This file contains a 4-bit incrementer with parallel carry look-ahead.

See Section 14.46 for the event file "v-inc4.events".

7.7 TV-DEC-PASS

The decrement-pass unit is used in register-indirect modes to compute the effective address quickly. There are two possibilities for the effective memory address, depending on whether or not the pre-decrement mode is set. The decrement-pass unit either decrements the address given to it or else just passes it through. We implement this unit using a propagate-generate based decrementer. Using the carry input as a control line, we either subtract `1` or `0`.

The combination of the following two lemmas shows that the decrement-pass unit has the desired semantics.

Theorem. `DEC-PASS$VALUE`

```
(implies
 (and (dec-pass& netlist n)
      (not (zerop n))
      (equal (length a) n)
      (properp a))
 (equal (dual-eval 0 #i(dec-pass n) (cons c a) state netlist)
        (f$dec-pass c a)))
```

Theorem. `F$DEC-PASS=DEC-OR-PASS`

```
(implies
 (and (bvp v)
      (boolp dec)
      (not (equal (length v) 0)))
 (equal (f$dec-pass dec v) (if* dec (v-dec v) v)))
```

See Section 14.47 for the event file "tv-dec-pass.events".

7.8 REG

This file defines two families of scanned registers, one write-enabled and one not. For those unfamiliar with this terminology, a scanned register is one that operates "normally" as a simple n-bit register, but can also operate as a *shift register*. A shift register is a sequence of one-bit registers where at each clock tick, the so-called *scan input* is latched into the first one-bit register and each bit of state is moved into the next bit of state. A *write-enabled* scan register is one that has an extra `write-enable` input that must be on in order for new values to be stored in the state.

An example evaluation of our scanned-register generators is shown below. The `DEL4` primitive is a 4-nanosecond delay buffer. We add this to ensure that there is sufficient hysteresis in the system, i.e., to ensure that minor variances in propagation of the clock signal to some state-holding devices do not cause premature changes to the inputs of other state-holding devices. Also, buffers are added to deal with fan-out on the write-enable and test-enable inputs. When the load demands are sufficiently great, `B-BUF-PWR` buffers are used in place of the `B-BUF` buffers shown below.

```
(lisp-netlist (reg* 3))

 =

'(REG_3 (CLK TE TI D_0 D_1 D_2)
        (Q_0 Q_1 Q_2)
        ((TE-BUFFER (TE-BUF) B-BUF (TE))
         (TI-DEL (TI-BUF) DEL4 (TI))
         (G_0 (Q_0 QB_0) FD1S (D_0 CLK TI-BUF TE-BUF))
         (G_1 (Q_1 QB_1) FD1S (D_1 CLK Q_0 TE-BUF))
         (G_2 (Q_2 QB_2) FD1S (D_2 CLK Q_1 TE-BUF)))
        (G_0 G_1 G_2))
```

```
(lisp-netlist (we-reg* 3))

 =

'(WE-REG_3 (CLK WE TE TI D_0 D_1 D_2)
           (Q_0 Q_1 Q_2)
           ((WE-BUFFER (WE-BUF) B-BUF (WE))
            (TE-BUFFER (TE-BUF) B-BUF (TE))
            (TI-DEL (TI-BUF) DEL4 (TI))
```

```
 (G_0 (Q_0 QB_0) FD1SLP (D_0 CLK WE-BUF TI-BUF TE-BUF))
 (G_1 (Q_1 QB_1) FD1SLP (D_1 CLK WE-BUF Q_0 TE-BUF))
 (G_2 (Q_2 QB_2) FD1SLP (D_2 CLK WE-BUF Q_1 TE-BUF)))
(G_0 G_1 G_2))
```

See Section 14.48 for the event file "reg.events".

Chapter 8

SPECIFICATION AND IMPLEMENTATION OF THE ALU

The first section below provides a user-level specification of the ALU at the bit level. Specifications at the natural number and integer level may be found in Section 13.2. The remaining sections of this chapter develop the implementation of our ALU.

8.1 ALU-SPECS

This file contains the high-level specification of the ALU, `V-ALU`. This function returns a carry bit, an overflow bit, a zero bit, and a bit vector of which the top bit is the negative bit. This ALU performs 14 different operations, including add-with-carry, add, shifting operations, and so on.

`V-ALU` is the bit-vector specification of the ALU. There are two other specification functions for the ALU, to be found in the file "alu-interpretation.events" (see Section 13.2). One function is `V-ALU-NAT`, which describes the ALU from the perspective of natural number operations. The other is the analogous function for (twos complement) integers, `V-ALU-INT`.

There are a number of simple rewrite rules in this file that we find useful. For example, since the ALU is used for the pre-decrement and post-increment operations on addresses, the following rewrite rule is used to remove calls of the ALU specification in favor of incrementing and decrementing operations.

Lemma. `BV-V-ALU-ALU-INC-ALU-DEC`

```
(and
```

```
(equal (bv (v-alu c a b (alu-inc-op)))
       (v-inc a))
(equal (bv (v-alu c a b (alu-dec-op)))
       (v-dec a)))
```

See Section 14.49 for the event file "alu-specs.events".

8.2 PRE-ALU

The ALU is essentially a tree with 32 cells at the leaves. Each of these cells performs the operation called for by the op-code. In fact, the cells have inputs that are controlled by so-called mode, propagate, and generate inputs, all of which are derived from the op-code together with two other control inputs, the zero input and the swap input.

This file defines the decoding logic necessary to generate the aforementioned mode, propagate, and generate inputs for the ALU cells. The functions `DECODE-GEN`, `DECODE-PROP`, and `DECODE-MODE` use random logic that has been designed to implement the table in this event file, as discussed in the next section, 8.3. It is interesting to note that these modules are defined by specifying their logic and using the `DEFN-TO-MODULE` macro (see Section 2.6) to generate the HDL descriptions automatically from these specifications. Here we also define a function `CARRY-IN-HELP` that provides the carry input for the entire ALU.

Further explanations of the ALU implementation may be found in the next section.

See Section 14.50 for the event file "pre-alu.events".

8.3 TV-ALU-HELP

Let us now provide a more high-level view of the definition of the ALU. The complete ALU is composed of a number of modules: `CARRY-OUT-HELP`, `TV-ALU-HELP`, `T-CARRY`, `SHIFT-OR-BUF`, `OVERFLOW-HELP`, and `V-ZEROP`. There are also certain parts of the ALU that are not included in the `CORE-ALU*` module because their computation is pipelined. These are the modules `CARRY-IN-HELP`, `DECODE-MODE`, `DECODE-PROP`, and `DECODE-GEN` discussed in the preceding section.

The heart of the ALU is the module `TV-ALU-HELP`, which is a propagate-generate ALU built in the style of a propagate-generate adder such as the specification `TV-ADDER` described in Section 7.1. However, instead of a 3-way exclusive OR to compute the sum at the leaves, we define `ALU-CELL` to do the appropriate computation at the leaves depending on the op-code. Note that since this module generator is tree-based, we do not have any macros to generate the HDL description automatically; hence the function `TV-ALU-HELP*` is hand-coded.

In order to make sense of this `ALU-CELL` module, let us consider the following portion of a table taken from a comment in the preceding section's event file. The "mode" bit indicates whether or not the carry-in is to be used.

```
FM9001   Mode  Prop  Gen C-in  C-out  Ovfl  Comment
opcode         aab   aab
3210            n     nn

0011     t     tft   ftt   f   co     *     Add

1011     f     tff   fft   x   f      f     XOR
```

Let us start with the second of the two rows displayed here. The XOR instruction has op-code `1011`, and since XOR is *not* an arithmetic operation (add, subtract, increment, or decrement), the mode flag is set to `f`. The basic operation of the `ALU-CELL` is the `B-EQUV3` primitive applied to the propagate, generate, and mode bits. These propagate and generate bits are produced respectively by the `P-CELL` and `G-CELL` modules defined below.

The basic motivation for our approach is that we need to produce propagate and generate bits for the arithmetic operations, and fortuitously we can compute the output from these two bits and the mode/carry bit, uniformly for all arithmetic op-codes, by using the `B-EQUV3` primitive. For the non-arithmetic op-codes these propagate and generate bits are computed so that the `B-EQUV3` gate gives the correct answer. Let us turn now to how these bits are computed.

Continuing with the XOR row displayed above, we note that the propagate bit is simply equal to the `a` input, since that is the only one marked with a `t`. However, for the Add operation described on the first row, the propagate bit is the logical OR of the `a` and `b` bits. More generally, the propagate bit is the logical OR of those of the `a`, negated `a`, and `b` bits that are marked with a `t` in the table above. This fact is illustrated by the following theorem, which the prover can easily check.

```
(equal
 (p-cell a an b pa pan pb)
 (or (and a pa)
     (and an pan)
     (and b pb)))
```

Actually, `P-CELL` is implemented using NAND gates, for efficiency reasons.

The generate bit is obtained similarly, but negated logic is used.

```
(equal (g-cell a an bn ga gan gbn)
       (not (or (and a ga)
```

```
                  (and an gan)
                  (and bn gbn))))
```

The generate field from the XOR line in the table above is `fft`, where the `t` bit is the `bn` (negated `b`) value, and hence the value returned by `G-CELL` is simply `b`. For the `Add` line, the generate field is `ftt`, so the result is the AND of `a` and `b`.

Finally let us turn to `ALU-CELL`. Its arguments are a carry input `c` resulting from the propagate-generate carry look-ahead tree; inputs `a` and `b` which are two corresponding bits from the input bit vectors; and a list `mpg` that contains the mode bit, the three `Prop` bits, and the three `Gen` bits from the table that we have been discussing.

Definition.

```
(alu-cell c a b mpg)
=
;; a and b are the inputs, which are single bits from two
;; input bit vectors for the ALU.  The carry bit c is either
;; generated by  CARRY-IN-HELP from the external input carry,
;; but only for the least significant bit of the ALU, or else
;; is generated by the propagate-generate carry look-ahead logic.
(let (;; First come the bits from the 'Gen' column, right to left,
      ;; which are produced from the op-code by  DECODE-GEN.
      (gbn (car mpg))
      (gan (cadr mpg))
      (ga  (caddr mpg))
      ;; Then come the bits from the 'Prop' column, right to left,
      ;; which are produced from the op-code by  DECODE-PROP.
      (pb  (cadddr mpg))
      (pan (caddddr mpg))
      (pa  (cadddddr mpg))
      ;; Finally comes the mode bit from the table,
      ;; which is produced from the op-code by  DECODE-MODE.
      ;; Essentially, m is true if and only if we are doing
      ;; an add, subtract, increment, or decrement operation.
      (m   (caddddddr mpg)))
  (let ((an (b-not a))
        (bn (b-not b)))
    (let ((p (p-cell a an b pa pan pb))
          (g (g-cell a an bn ga gan gbn))
          (mc (b-nand c m)))
      (list p g (b-equv3 mc p g)))))
```

The call of B-EQUV3 in the definition above can be simplified depending on the mode flag, m, as the following theorems demonstrate. Note that perhaps contrary to its name, B-EQUV3 has the property that (B-EQUV3 X Y Z) is equal to (NOT (B-EQUV X (B-EQUV Y Z))).

```
;; XOR case
(equal (b-equv3 (b-nand c f) a (not (not b)))
       (xor a b))

;; Add case
(equal (b-equv3 (b-nand c t) (or a b) (and a b))
       (b-xor3 c a b))
```

See Section 14.51 for the event file "tv-alu-help.events".

8.4 POST-ALU

This file implements part of the ALU's functionality. The modules CARRY-OUT--HELP, OVERFLOW-HELP, and TV-SHIFT-OR-BUF are connected to the output of the module TV-ALU-HELP. The module TV-SHIFT-OR-BUF performs right shift operations if necessary; otherwise, it just buffers its input.

CARRY-OUT-HELP produces the carry output of the complete ALU. For instance, if the zero input is set then the carry output is F regardless of anything else. For another example, the right shift instructions, the carry output is the least significant bit of the a input. Actually, random logic implements this module in conformance with the table at the top of the events file for Section 8.2, using the DEFN-TO-MODULE macro (cf. Section 2.6). The definition of OVERFLOW-HELP is analogous to that of CARRY-OUT-HELP.

See Section 14.52 for the event file "post-alu.events".

8.5 CORE-ALU

This file contains the proof that the ALU's top-level specification is implemented in two steps. First, CARRY-IN-HELP and MPG do the appropriate preprocessing (as indicated by the bindings in the LET form below); then, CORE-ALU correctly computes the ALU result. The following lemma summarizes this two-step process. It may be viewed as a composition of several lemmas that appear in this and preceding files, especially the "$VALUE" lemmas for CORE-ALU, CARRY-IN-HELP, and MPG as well as the lemma CORE-ALU-IS-V-ALU that relates a structural description CORE-ALU of the ALU with a high-level specification V-ALU of the ALU.

Theorem. `CORE-ALU-TOP`

```
(let ((c (car (dual-eval 0
                         'carry-in-help
                         (cons carry-in (cons f op))
                         state
                         netlist)))
      (mpg (dual-eval 0 'mpg (cons zero (cons f op))
                      state netlist)))
  (implies
   (and (core-alu& netlist tree)
        (carry-in-help& netlist)
        (mpg& netlist)
        (equal (length a) (tree-size tree))
        (bv2p a b)
        (geq (length a) 3)
        (not zero)
        (bvp op)
        (boolp carry-in)
        (equal (length op) 4))
   (equal (dual-eval
           0
           (index 'core-alu (tree-number tree))
           (cons c (append a (append b (cons zero
                                             (append mpg op)))))
           state
           netlist)
          (v-alu carry-in a b op))))
```

Another lemma in the file, `CORE-ALU-WORKS-FOR-ZERO-CASE`, describes what happens when the zero input is set. This completes our introduction to this file, except to note for those who are interested in theorem-proving details that the lemma `CASES-ON-A-4-BIT-BVP` is used as a sneaky case-splitting hint.

See Section 14.53 for the event file "core-alu.events".

Chapter 9

SPECIFICATION OF THE FM9001

The following section provides the specification of the FM9001 at the user level. The other section in this chapter is merely a presentation of a simple assembler used for testing the specification.

9.1 FM9001-SPEC

The FM9001 interpreter is defined in the standard manner: as a recursive function that repeatedly "applies" a "single-step" function.

Definition.

```
(FM9001 state n)
 =
(if (zerop n)
    state
  (FM9001 (FM9001-step state (nat-to-v 15 (reg-size)))
          (sub1 n)))
```

The definition of the "single-stepper" `FM9001-STEP` makes use of a sequence of auxiliary functions. The particular fields of an instruction word do not concern us at this point; they will be addressed once we complete our top-down high-level view of the stepper.

The `STATE` parameter to `FM9001-STEP` will be examined further below, and `PC-REG` may be thought of as the list `(LIST T T T T)` representing register 15. Let us describe the high-level structure of this "calling tree" before looking at each such function in some detail.

(FM9001-step state pc-reg)
-->
(FM9001-fetch *register-file-0 flags memory* pc-reg)
-->
(FM9001-operand-a *register-file-1 flags memory instruction*)
-->
(FM9001-operand-b *register-file-2 flags memory instruction operand-a*)
-->
(FM9001-alu-operation *register-file-3 flags memory instruction operand-a operand-b b-address*)
=
(list (list *new-register-file new-flags*) *new-memory*)

The definition of FM9001-STEP gives a clue to the structure of the STATE parameter of the interpreter function shown above. Namely: the state is of the form (LIST P-STATE MEM), where P-STATE contains fields for the register file and the flags, and MEM is the processor memory. The register file, flags, and memory, together with the index of the program counter register (typically 15) expressed as a bit vector, are then handed off to the function FM9001-FETCH, which implements the fetch-execute cycle.

Definition.

```
(FM9001-step state pc-reg)
 =
(let ((p-state (car state))
      (mem     (cadr state)))
  (FM9001-fetch (regs p-state) (flags p-state) mem pc-reg))
```

The function FM9001-FETCH implements the fetch-execute cycle by obtaining the program counter PC, the current instruction INS, and the new register file obtained by incrementing the program counter, and then calling the function FM9001-OPERAND-A that we describe next.

Definition.

```
(FM9001-FETCH regs flags mem pc-reg)
 =
(let ((pc (read-mem pc-reg regs)))
  (let ((ins (read-mem pc mem)))
    (let ((pc+1 (v-inc pc)))
      (let ((new-regs (write-mem pc-reg regs pc+1)))
        (FM9001-operand-a new-regs flags mem ins)))))
```

The function FM9001-OPERAND-A obtains the first operand, OPERAND-A, either as an immediate value directly from the instruction word, as a value directly from register A, or else by reading from the memory using the address stored in register A (possibly pre-decremented). OPERAND-A is an argument of the call to the next function FM9001-OPERAND-B, which is given a new register file if the mode calls for pre-decrementing or post-incrementing of the first operand register.

Definition.

```
(fm9001-operand-a regs flags mem ins)
 =
(let ((a-immediate-p (a-immediate-p ins))
      (a-immediate   (sign-extend (a-immediate ins) 32))
      (mode-a        (mode-a ins))
      (rn-a          (rn-a  ins)))
  (let ((reg (read-mem rn-a regs)))
    (let ((reg- (v-dec reg))
          (reg+ (v-inc reg)))
      (let ((operand-a (if a-immediate-p
                           a-immediate
                         (if (reg-direct-p mode-a)
                             reg
                           (if (pre-dec-p mode-a)
                               (read-mem reg- mem)
                             (read-mem reg mem))))))
        (let ((new-regs (if a-immediate-p
                            regs
                          (if (pre-dec-p mode-a)
                              (write-mem rn-a regs reg-)
                            (if (post-inc-p mode-a)
                                (write-mem rn-a regs reg+)
                              regs)))))

          (FM9001-operand-b new-regs flags mem ins operand-a))))))
```

The function FM9001-OPERAND-B is defined analogously to the function above except that there is no immediate operand mode. Thus, the second operand, OPERAND-B, is obtained and passed to the next function, FM9001-ALU-OPERATION, along with a register file, in which the second operand register has been (possibly) updated. The value stored in the second operand register is saved (after pre-decrementing if appropriate) and passed to the ALU as the additional argument B-ADDRESS.

Definition.

```
(fm9001-operand-b regs flags mem ins operand-a)
 =
(let ((mode-b (mode-b ins))
      (rn-b   (rn-b  ins)))
  (let ((reg (read-mem rn-b regs)))
    (let ((reg- (v-dec reg))
          (reg+ (v-inc reg)))
      (let ((b-address (if (pre-dec-p mode-b)
                           reg-
                         reg)))
        (let ((operand-b (if (reg-direct-p mode-b)
                             reg
                           (read-mem b-address mem)))
              (new-regs (if (pre-dec-p mode-b)
                            (write-mem rn-b regs reg-)
                          (if (post-inc-p mode-b)
                              (write-mem rn-b regs reg+)
                            regs))))

          (FM9001-alu-operation
           new-regs flags mem ins
           operand-a operand-b b-address))))))
```

Finally, the function `FM9001-ALU-OPERATION` is called to return the new state, i.e., the new register file, the new flags, and the new memory. The new register file `NEW-REGS` is obtained by writing the bit vector returned by the ALU into register B, assuming that the flags are set appropriately for the given instruction (as indicated by `STOREP`) and that the mode for register B is "direct." The new flags `NEW-FLAGS` are set using the flags returned by the ALU, but only for the flags that the instruction requires to be updated. Finally, the new memory is obtained by writing the ALU's bit-vector result to the memory at the appropriate location obtained from register B, assuming that (once again) `STOREP` is set, and assuming that the mode for register B is register-indirect.

Definition.

```
(fm9001-alu-operation regs flags mem ins
                      operand-a operand-b b-address)
 =
(let ((op-code   (op-code ins))
      (store-cc  (store-cc ins))
      (set-flags (set-flags ins))
      (mode-b    (mode-b   ins))
      (rn-b      (rn-b    ins)))
  (let ((cvzbv  (v-alu (c-flag flags) operand-a operand-b op-code))
```

```
        (storep (store-resultp store-cc flags)))
    (let ((bv (bv cvzbv)))
      (let ((new-regs   (if (and storep (reg-direct-p mode-b))
                            (write-mem rn-b regs bv)
                          regs))
            (new-flags  (update-flags flags set-flags cvzbv))
            (new-mem    (if (and storep (not (reg-direct-p mode-b)))
                            (write-mem b-address mem bv)
                          mem)))

        (list (list new-regs new-flags) new-mem)))))
```

There are two more interpreter functions defined in this file. The function `FM9001-INTERPRETER` takes the program counter register as an argument. The function `FM9001-INTR` goes even further by allowing an oracle to supply the program counter for each instruction. The file concludes with some simple lemmas.

See Section 14.54 for the event file "fm9001-spec.events".

9.2 ASM-FM9001

This file contains a quick-and-dirty assembler, together with example programs at the end of the file. It produces a list that is readily converted into a memory for the behavioral specification function `FM9001`.

This assembler is not used in the proof, and nothing is proved about it. It is simply a convenient tool for testing the FM9001 specification.

See Section 14.55 for the event file "asm-fm9001.events".

Chapter 10

FM9001 HARDWARE MODULES

In this chapter we implement the bulk of the logic for the FM9001 other than the ALU.

10.1 STORE-RESULTP

This file defines hardware implementing the specification function `STORE-RESULTP`, introduced in Section 9.1, that determines when to store the answer produced by the ALU. More precisely, the ALU always produces a bit vector and some flags; this hardware decides only whether or not the bit-vector result should be stored.

See Section 14.56 for the event file "store-resultp.events".

10.2 CONTROL-MODULES

The file "control-modules.events" defines several hardware functions: a unary operation-code recognizer, a register-mode recognizer, opcode selection logic, fanout modules, decoders and encoders. These are defined in a similar manner to many previously defined modules. These are submodules of the main control logic defined in the next section.

See Section 14.57 for the event file "control-modules-events.events".

10.3 CONTROL

In this section we describe the control logic, which is perhaps the most complicated part of the FM9001 definition and proof. Most of the control logic is generated automatically, and so are most of the proof events. This section depends heavily on macros defined in the corresponding Lisp file "control.lisp", which is mentioned in Section 2.11 but which we document here.

The goal of this file is to define the module `NEXT-CNTL-STATE`, which takes a 5-bit major control state (as defined below) and some other inputs and returns a 40-bit control vector. That control vector contains 5 bits comprising the "next major control state" together with 35 additional control fields. The 40-bit control vector produced by the module `NEXT-CNTL-STATE` controls the entire FM9001 microarchitecture and the memory protocol interface. That is, this function is literally the control logic for the FM9001. The microarchitecture of the FM9001 is shown in Figure 10.1. In this conventional engineering diagram, we see in the upper half a description of the "data path" and in the lower half a descirption of the "control logic." Obviously, this diagram is an abstraction because as drawn the control logic is not even "connected" to the data path. The 40-bit vector contained in the `CNTL-STATE` register literally controls the machine. The "function" `NEXT-CNTL-STATE` produces a new 40-bit control vector for each clock cycle, as a function of the previous values of these seven quantities: `CNTL-STATE`, `FLAGS`, `RESET-`, `HOLD-`, `DTACK-`, `PC-REG`, and the instruction register `I-REG`. The upper half of Figure 10.1 illustrates the principal data paths. The upper righthand corner shows the bi-directional external memory databus, and just below it is the external address bus. The `REGISTER-FILE`, `A-REG`, `B-REG`, `I-REG`, `DATA-OUT`, and `ADDR-OUT` are state-holding registers. The other blocks, e.g., the `ALU` are combinational logic.

An instruction is executed by the microarchitecture by reading an instruction from the `DATA-BUS` (which is connected to the external memory) and storing it into `I-REG`. The instruction is then decoded and operands are fetched and then stored into `A-REG` and `B-REG`, as necessary. This decoding and storing of operands may require several clock cycles. The result of the `ALU` computation is stored as specified by the instruction.

The `NEXT-CNTL-STATE` module is defined in the file "control.events" through the definition of several major subpieces: `CONTROL-LET`, `DECODE-5` (defined in "control-modules.events"), `NEXT-STATE`, and `CV`. A diagram of the `NEXT-CNTL-STATE` module is shown in Figure 10.2. We will return to this module at the end of this section, after we present the major subpieces.

Before we are able to define the various modules that comprise `NEXT-CNTL-STATE` we define a set of tables and functions that specify what the `NEXT-CNTL-STATE` module is to do. The complicated part of this file begins with the part labeled in the event file as "Macro Execution," which contains the following three macro calls.

```
(define-control-states)

(define-control-state-accessors)
```

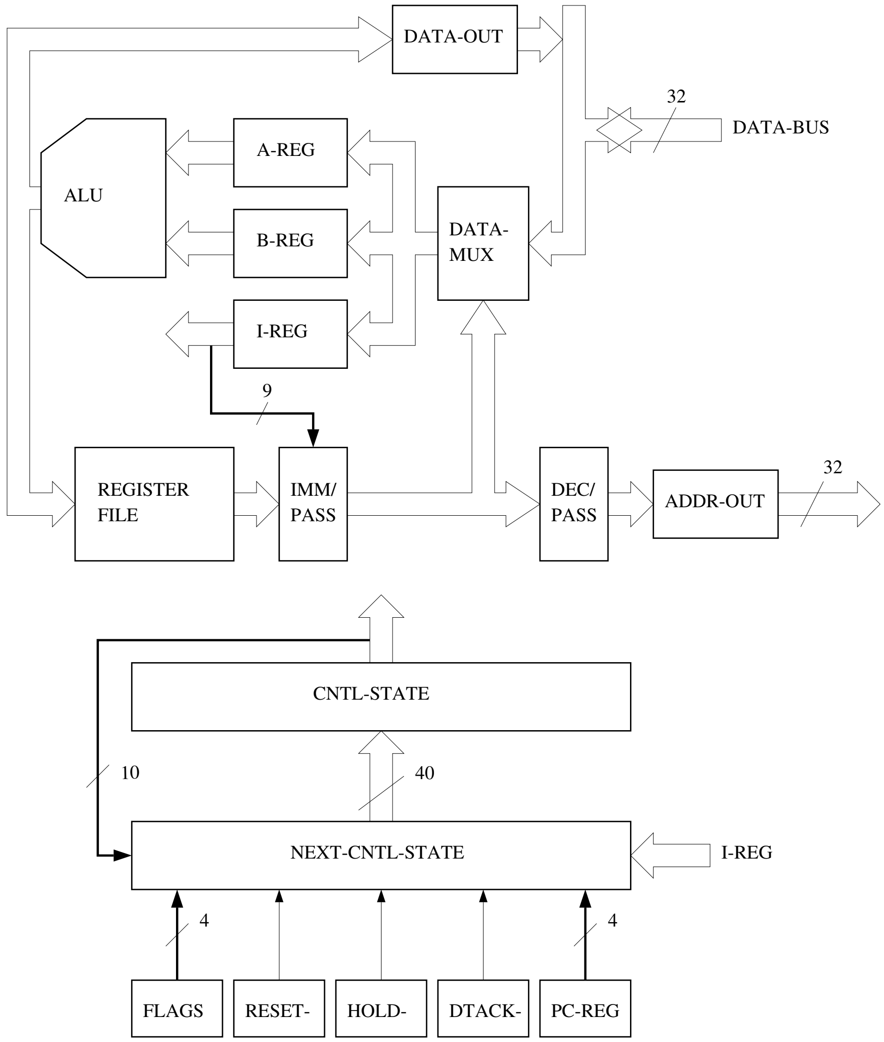


Figure 10.1: Block Diagram of the FM9001

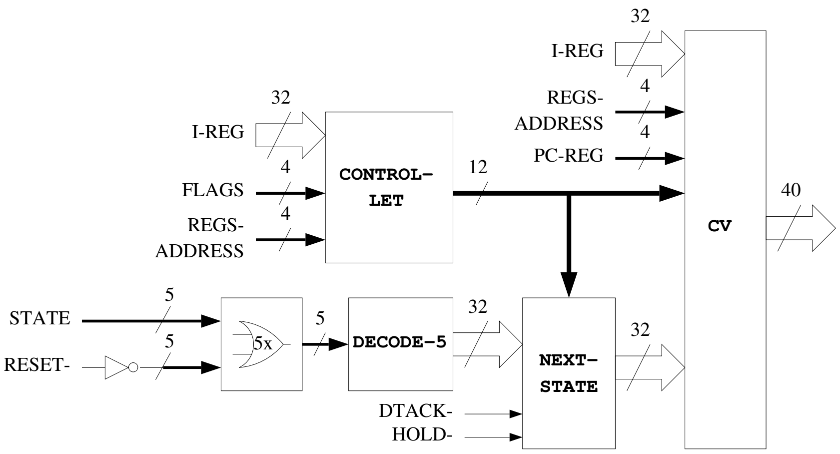


Figure 10.2: Diagram of the `NEXT-CNTL-STATE` Module

```
(define-control-vector-functions)
```

These macros, which are defined in the file "control.lisp", generate events for the control logic using a sort of microcode-style description of the control logic. Let us look at this process in more detail.

The operation of the `FM9001` is controlled by a 40-bit *control vector*. Most of the bits of this control vector directly control internal latches and selectors, the ALU, the register file, and the memory interface. For instance, one bit is a write-enable control bit for the address output latch. Especially noteworthy is a five-bit subpart of the control vector that defines the *major control state* for the machine. These 32 major control states (actually 29, since three are "unused") are used to define a finite-state machine, whose transitions depend not only on the current state but also on other inputs, including the instruction register, the data acknowledgement input, and the hold input.

The Lisp constant `*CONTROL-STATES*` defined in "control.lisp" gives symbolic names to the 32 major control states.

```
(defconstant
  *control-states*
  '((fetch0  . #v00000)  ;; setting up for instruction fetch
    (fetch1  . #v00001)  ;; checking for hold
    (fetch2  . #v00010)  ;; increment pc
    (fetch3  . #v00011)  ;; wait for data ack., read instruction

    (decode  . #v00100)  ;; decode the instruction
    (rega    . #v00101)  ;; fetch operand A from register file
    (regb    . #v00110)  ;; fetch operand B from register file
```

```
(update  . #v00111)  ;; update register file with answer

(reada0  . #v01000)  ;; setting up for operand A memory read
(reada1  . #v01001)  ;; do pre-decrement or post-increment, if necessary
(reada2  . #v01010)  ;; store the result of pre-decrement or post-increment
(reada3  . #v01011)  ;; wait for data ack., read operand A data

(readb0  . #v01100)  ;; setting up for operand B memory read
(readb1  . #v01101)  ;; do pre-decrement or post-increment, if necessary
(readb2  . #v01110)  ;; store the result of pre-decrement or post-increment
(readb3  . #v01111)  ;; wait for data ack., read operand B data

(write0  . #v10000)  ;; setting up for the instruction result write
(write1  . #v10001)  ;; may do pre-dec or post-inc, if skipped above
(write2  . #v10010)  ;; turn memory strobe on
(write3  . #v10011)  ;; wait for data acknowledgement

(sefa0   . #v10100)  ;; get register A from register file
(sefa1   . #v10101)  ;; side effect operand A register, if necessary
(sefb0   . #v10110)  ;; get register B from register file
(sefb1   . #v10111)  ;; side effect operand B register, if necessary

(hold0   . #v11000)  ;; hold state
(hold1   . #v11001)  ;; exiting hold state
(v11010  . #v11010)  ;; unused state
(v11011  . #v11011)  ;; unused state

(reset0  . #v11100)  ;; reset state, make ALU generate a zero
(reset1  . #v11101)  ;; write zero in address latch, data output, etc.
(reset2  . #v11110)  ;; zero all 16 registers in register file
(v11111  . #v11111)  ;; unused state
))
```

The macro DEFINE-CONTROL-STATES is used to create a number of Nqthm events that "implement" the table above in the Nqthm logic. For each major control state it submits six events which define two constants, such as the six shown here for the first state.

```
(DEFN V_FETCH0 NIL (LIST F F F F F))
(DISABLE V_FETCH0)
(DISABLE *1*V_FETCH0)

(DEFN N_FETCH0 NIL (V-TO-NAT (LIST F F F F F)))
(DISABLE N_FETCH0)
(DISABLE *1*N_FETCH0)
```

Next, a DEFTHEORY event defines a theory VECTOR-STATE-THEORY that contains the names of all the "V-" functions just defined, such as V-FETCH0 from the display above, and then a second DEFTHEORY event defines a theory NATURAL-STATE-THEORY that contains the names of all the "N-" functions just defined, such as N-FETCH0 from the display above. These two events are followed by a lemma BVP-LENGTH--STATE-VECTORS that defines two rewrite rules for each major control state, such as the following two for the FETCH0 state.

```
(EQUAL (LENGTH (V_FETCH0)) 5)
(BVP (V_FETCH0))
```

Finally, the lemma just proved is disabled.

The macro DEFINE-CONTROL-STATE-ACCESSORS, also defined in "control.lisp" and used in the file "control.events", defines accessors for the control state, and concludes with a DEFTHEORY event that creates a theory containing these accessors. It macroexpands as follows (many lines omitted, as shown):

```
(PROGN
  (DEFN RW- (CNTL-STATE) (NTH 0 CNTL-STATE))
  (DISABLE RW-)
  (DEFN STROBE- (CNTL-STATE) (NTH 1 CNTL-STATE))
  (DISABLE STROBE-)
  ;; ... and so on for fields 2 through 14
  (DEFN ALU-ZERO (CNTL-STATE) (NTH 15 CNTL-STATE))
  (DISABLE ALU-ZERO)
  (DEFN STATE (CNTL-STATE) (SUBRANGE CNTL-STATE 16 20))
  (DISABLE STATE)
  ;; ... and so on
  (DEFN ALU-MPG (CNTL-STATE) (SUBRANGE CNTL-STATE 33 39))
  (DISABLE ALU-MPG)
  (DEFTHEORY CONTROL-STATE-ACCESSOR-THEORY
      (RW- STROBE- HDACK- WE-REGS WE-A-REG WE-B-REG WE-I-REG
           WE-DATA-OUT WE-ADDR-OUT WE-HOLD- WE-PC-REG DATA-IN-SELECT
           DEC-ADDR-OUT SELECT-IMMEDIATE ALU-C ALU-ZERO STATE
           WE-FLAGS REGS-ADDRESS ALU-OP ALU-MPG)))
```

Finally consider the third of the three macros mentioned above, DEFINE--CONTROL-VECTOR-FUNCTIONS. It is used to define functions that are used to transition to the major control states defined in the constant *CONTROL-STATES* displayed above. Before we give an example of such a definition, let us look at two more constants from the Lisp file "control.lisp".

Recall that the major control state is just 5 bits of the 40-bit control vector. Our finite state machine will not only take us from a given 5-bit major control state to a new state, but will also supply values for the other fields in the 40-bit control vector. The following table supplies the default values for fields of the control vector. That is, below we describe a table `*STATE-TABLE*` that only displays the fields that take on other than the default values for all 40 bits of the control vector. We describe each of these fields later in this section.

```
(defconstant
  *control-template*
  ;; Line       Type (Bit/Vector)  Default
  '((rw-                b           t)
    (strobe-            b           t)
    (hdack-             b           t)
    (we-regs            b           f)
    (we-a-reg           b           f)
    (we-b-reg           b           f)
    (we-i-reg           b           f)
    (we-data-out        b           f)
    (we-addr-out        b           f)
    (we-hold-           b           f)
    (we-pc-reg          b           f)
    (data-in-select     b           f)
    (dec-addr-out       b           f)
    (select-immediate   b           f)
    (alu-c              b           (carry-in-help
                                      (cons c (cons f op-code))))
    (alu-zero           b           f)
    (state              5           #v00000)
    (we-flags           4           (make-list 4 f))
    (regs-address       4           pc-reg)
    (alu-op             4           op-code)
    (alu-mpg            7           (mpg
                                     (cons f
                                          (cons f op-code))))))
```

Now we can describe the transitions from each of the major control states. These are also given as state transition diagrams in Figures 10.3 through 10.9. These state diagrams are described symbolically by the constant `*STATE-TABLE*`, which is defined in the file "control.lisp". Each entry in this table indicates the current major control state (as named in the constant above `*CONTROL-STATES*`), the next major control state, and zero or more specifications of settings for the other fields in the control state. To repeat, each entry in `*STATE-TABLE*` specifies for each major control state what the next 40-bit control state is to be (of which five bits indicate the next major control state). Some of these specifications are just field names specified by the constant `*CONTROL-TEMPLATE*`, indicating that we are to use the negation of the default specified by that constant. This negation approach does not work for fields of bits. There are entries of the form `(field term)`, indicating that `term` is to be the non-default value for `field`. Here then is a portion of our specification of the control algorithm for the FM9001.

```
(defconstant
  *state-table*
  '((fetch0 fetch1 we-addr-out we-a-reg (regs-address pc-reg)
            (rw- rw-) we-hold-
            alu-zero
            (alu-op (alu-inc-op))
            (alu-c (carry-in-help (cons c (cons t (alu-inc-op)))))
            (alu-mpg (mpg (cons t (cons f (alu-inc-op))))))

    (fetch1 (if hold- fetch2 hold0))

    (fetch2 fetch3 strobe-
            we-regs (regs-address pc-reg)
            (alu-c (carry-in-help (cons c (cons f (alu-inc-op)))))
            (alu-op (alu-inc-op))
            (alu-mpg (mpg (cons f (cons f (alu-inc-op))))))
```

..... *many entries omitted* .....

```
    (reset2 (if all-t-regs-address fetch0 reset2)
            we-regs
            alu-zero
            (alu-op (alu-inc-op))
            (alu-c (carry-in-help (cons c (cons t (alu-inc-op)))))
            (alu-mpg (mpg (cons t (cons f (alu-inc-op)))))
            (regs-address (v-inc regs-address)))

    (v11111 reset0)))
```

Let us describe a couple of the transitions represented by this table. Consider the initial major state, `fetch0`. The first entry in this table tells us that the next state will be `fetch1`, and that the remaining 35 fields of the control state vector will take on their default values *except* for the ones indicated in the entry in the table. So, for example, we see that `we-addr-out` and `we-a-reg` obtain the negations of their defaults shown in the constant `*CONTROL-STATES*` given earlier, while `regs-address` will attain the current value of `pc-reg` and `rw-` will retain its current value ...The entry for `reset2` is interesting because the next major control state can be either `fetch0` or `reset2`, depending on the current value of `all-t-regs-address`. This branching aspect is illustrated at the bottom of Figure 10.9. From state `reset2`, if `REGS-ADDRESS` is 15 (actually a four-bit vector containing `T`'s), then we go to major state `fetch0`; otherwise, we go right back to major state `reset2`.

Remark. The above description is accurate for the finite state machine we have described. However, in the context of the entire chip one needs to be careful when reading `*STATE-TABLE*`. For instance, consider our `fetch0` example above. In particular, consider the entry `WE-ADDR-OUT`. Since the default for this field is `F`, this entry says that `WE-ADDR-OUT` is to take on the value `T` in the next control state. `WE-ADDR-OUT` controls when the 32-bit address output latch is to latch in a new value. Note however that the address latch is not affected until the following major control state (`fetch2`). This is because each new control state, as specified by `*STATE-TABLE*`, is stored in a 40-bit register, thus the new control bits do not affect the FM9001 microarchitecture until the following major state. This "pipelining" of the control logic was done in order to enhance testability of the chip. By using the built-in register scan chain, it is possible to set the control vector to any 40-bit value for testing purposes. However, this approach does reduce the performance of the machine, because we are forced to use one clock cycle for instruction decoding.

Let us return to discussion of the macro `DEFINE-CONTROL-VECTOR-FUNCTIONS`. This macro generates a definition of a specification function which computes the next 40-bit control state for each major control state shown in `*STATE-TABLE*`. Thus, for each major control state the macro creates a definition of the transition function. Each generated definition takes the arguments shown below (note that `I-REG` holds the current instruction, which provides a lot of information), and computes the control vector for each major control state. This macro also creates a "destructuring" lemma describing the components of that next control state vector, and it disables both the definition and the lemma. It then disables both the definition and the lemma generated for each major control state. The macro `DEFINE-CONTROL-VECTOR-FUNCTIONS` also defines a couple of theories and proves that all of the next control vectors are bit vectors, and then disables that lemma.

The next major section of "control.events" defines the module `CONTROL-LET`, which can be seen in Figure 10.2. This function is a "helper function" that defines twelve values used in the computation of the next control vector. The definition of the `CONTROL-LET` module is completed by the two macro calls:

```
(module-predicate control-let*)

(module-netlist control-let*)
```

The logic for the module `NEXT-STATE` is synthesized from the `*STATE-TABLE*`, defined above. The output of the `NEXT-STATE` module is a 32-bit vector, of which only one bit is `T`, indicating what next major control state the machine should enter.

The module `CV` produces the 40-bit control vector. It does this by using the inputs from the `CONTROL-LET` and `NEXT-STATE` modules and the instruction register `I-REG`, the four-bit pointer to the register containing the program counter `PC-REG`, and the four-bit register `REGS-ADDRESS`, which is used to specify the register in the register file is to be accessed. The hardware module `CV` takes the 32-bit, one-hot encoding of the next major control state and other inputs to produce the next 40-bit control vector. Note that `CV` is a part of `NEXT-STATE`, which actually provides the 40-bit control vector.

We now describe each of the 40 bits of the control vector. These bits are computed in module `CV`, which is the last module of `NEXT-CNTL-STATE`. These bits are computed combinationally, and are stored each clock cycle in a 40-bit register. Thus, they do not affect the operation of the microarchitecture until the following clock cycle.

- `RW-` — memory read/write control
- `STROBE-` — memory strobe
- `HDACK-` — hold acknowledgement
- `WE-REGS` — enable write into the register file
- `WE-A-REG` — enable write into `A-REG`
- `WE-B-REG` — enable write into `B-REG`
- `WE-I-REG` — enable write into `I-REG`
- `WE-DATA-OUT` — enable write into `DATA-OUT`
- `WE-ADDR-OUT` — enable write into `ADDR-OUT`
- `WE-HOLD-` — enable write into the `HOLD-` input register
- `WE-PC-REG` — enable write into the `PC-REG` input register
- `DATA-IN-SELECT` — selects input source for `A-REG`, `B-REG`, and `I-REG`
- `DEC-ADDR-OUT` — selects whether to decrement value destined for `ADDR-OUT` register
- `SELECT-IMMEDIATE` — controls whether nine-bit immediate value is used
- `ALU-C` — input carry to the `ALU`
- `ALU-ZERO` — forces `ALU` output to all `F` values
- `STATE` — five-bit major control state
- `WE-FLAGS` — four-bit write enable for each of the `FLAGS`

- REGS-ADDRESS — four-bit value specifying register in register file to use for read or write
- ALU-OP — four-bit value specifying principal ALU operation
- ALU-MPG — seven-bit value specifying ALU operation

The module NEXT-CNTL-STATE glues together the four modules CONTROL-LET, DECODE-5, NEXT-STATE, and CV to produce the 40-bit control vector. This is shown in Figure 10.2. The logic for the entire contents of Figure 10.2 is purely combinational.

Perhaps it would be useful to trace through the major control states reached when a particular instruction is executed. First consider the decoding of an arbitrary instruction. The state names used correspond to those given earlier in the constant *CONTROL-STATE* presented earlier in this section. To execute an instruction, we always start in major control state FETCH0 (see the top circle in Figure 10.3), where we move the program counter address to the ADDR-OUT register in anticipation of reading an instruction from memory. We move the program counter address to the A-REG so we can increment this address. We also sample the HOLD- input, which is tested subsequently in state FETCH1. If a HOLD- is asserted, then we procede to major control state HOLD0. Otherwise, we proceed to state FETCH2, where we store the incremented program counter back into the register file and assert the external memory strobe line, STROBE-. We now proceed to state FETCH3, where we wait for DTACK- to be asserted. At the termination of state FETCH3, we load the instruction register I-REG with the value read from memory. We now have the instruction to be executed in I-REG.

We are now ready to "decode" the instruction, and one of six major control states is selected, depending upon the type of instruction. This is shown at the bottom of Figure 10.3. Notice that the righthand-most branch contains the next major control state destination FETCH0, which is a "no-op" instruction. The other five target major control states are picked depending on the nature of the instruction. These different states are selected depending upon whether the instruction is binary or unary, whether the A-operand is register-direct or register-indirect, and, in the case of memory-indirect addressing, whether the register specified by the A-operand should be pre-decremented. Figures 10.4 through Figures 10.8 contain the details for executing the various instructions. Figure 10.9 contains the HOLD-acknowledge control algorithm and the reset control algorithm.

See Section 14.58 for the event file "control-events.events".

10.4 REGFILE

A high-level view of the FM9001 register file is given in Figure 10.10. When the write-enable line WE is set, the DATA vector is latched into the register file at the

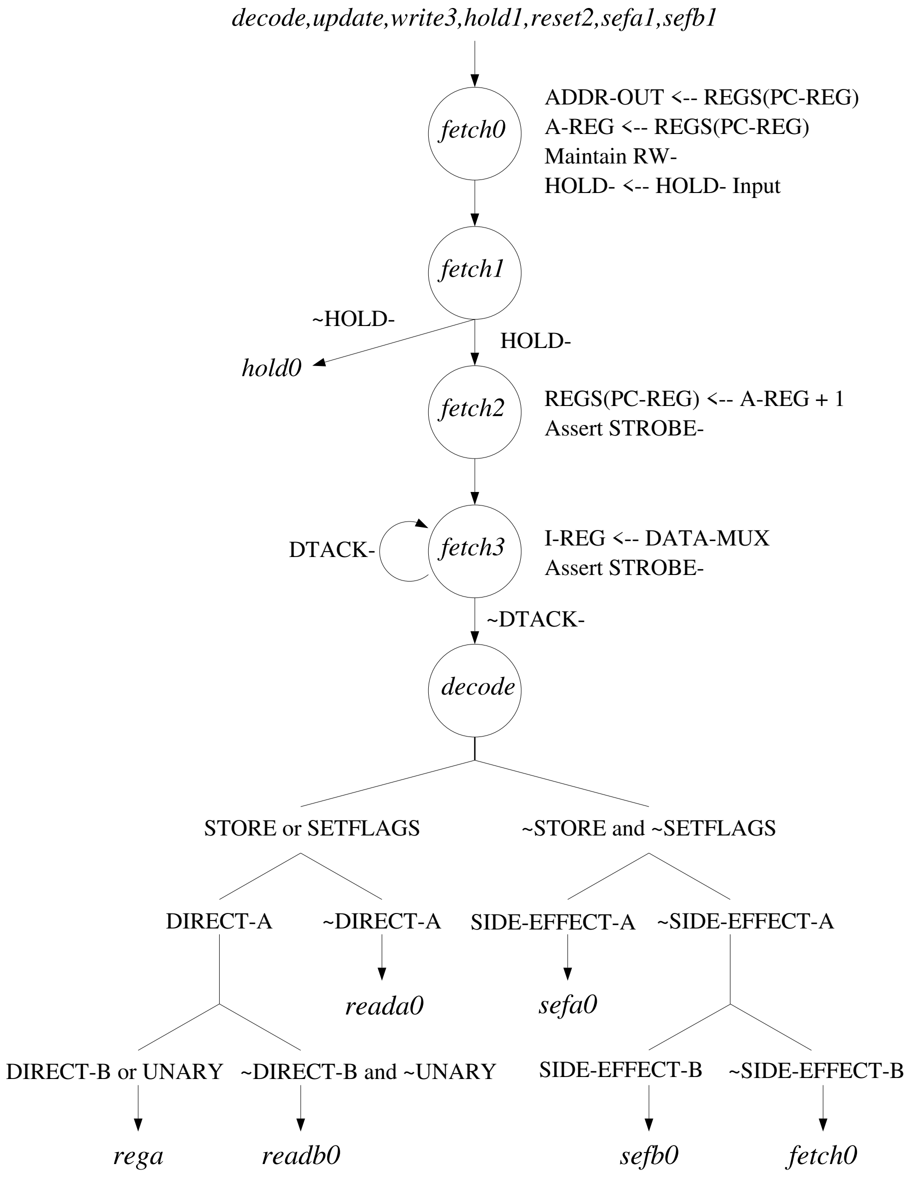


Figure 10.3: Major Control State Diagram 0

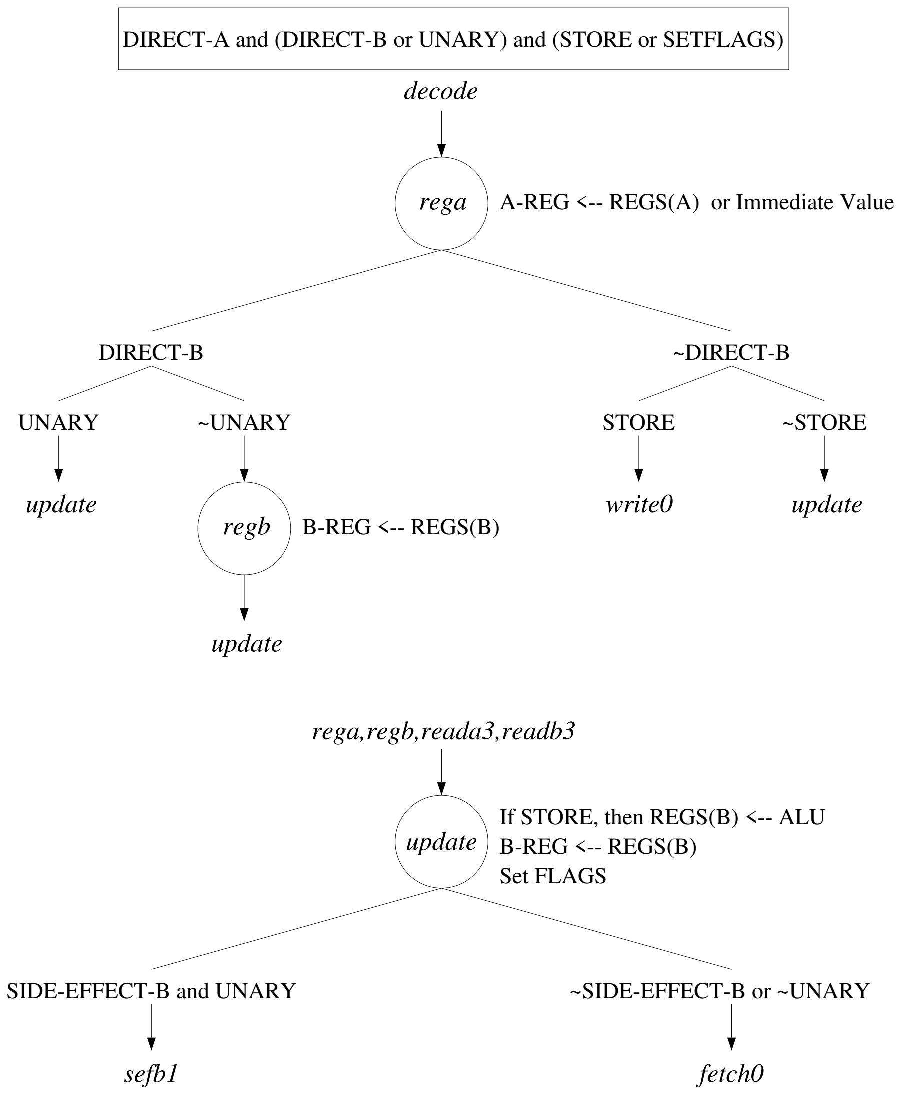


Figure 10.4: Major Control State Diagram 1

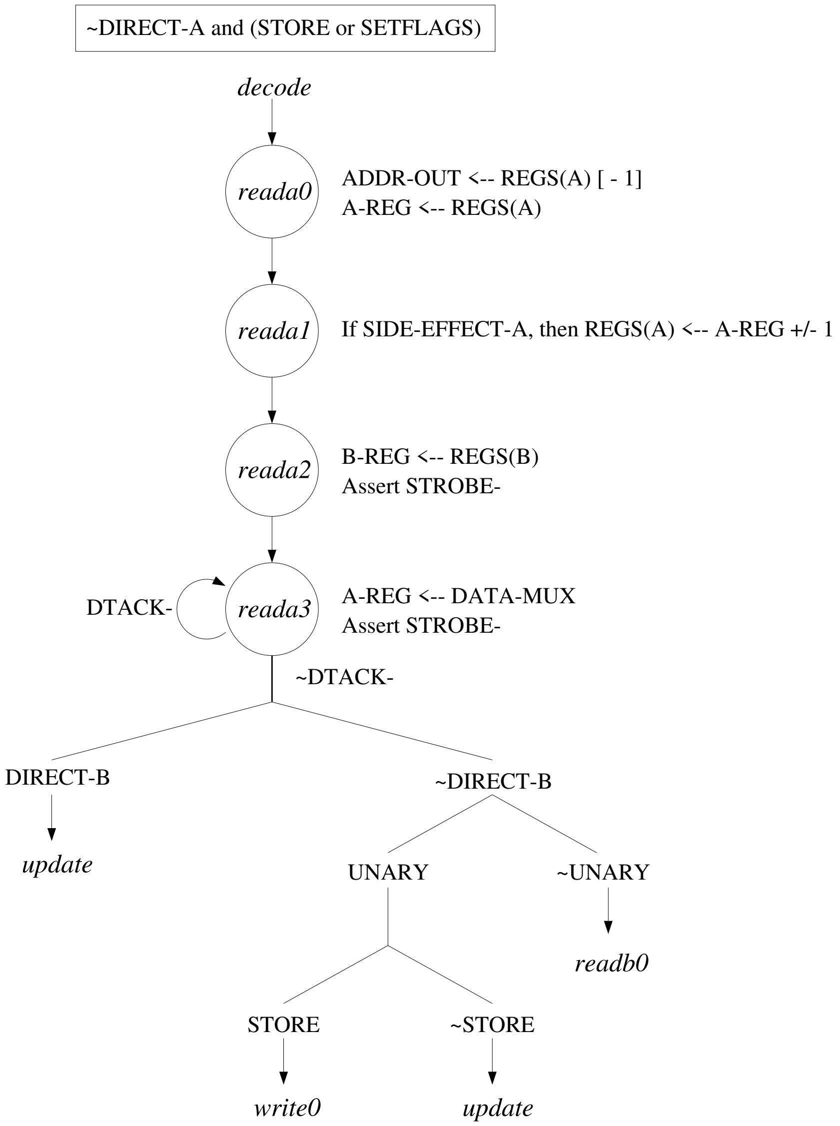


Figure 10.5: Major Control State Diagram 2

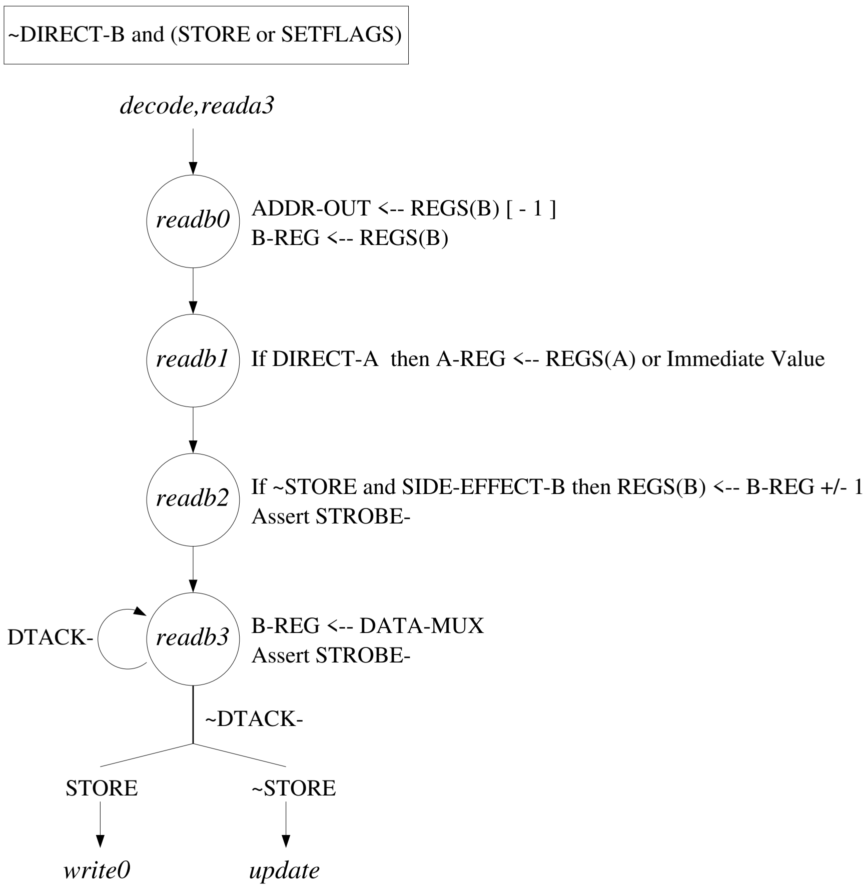


Figure 10.6: Major Control State Diagram 3

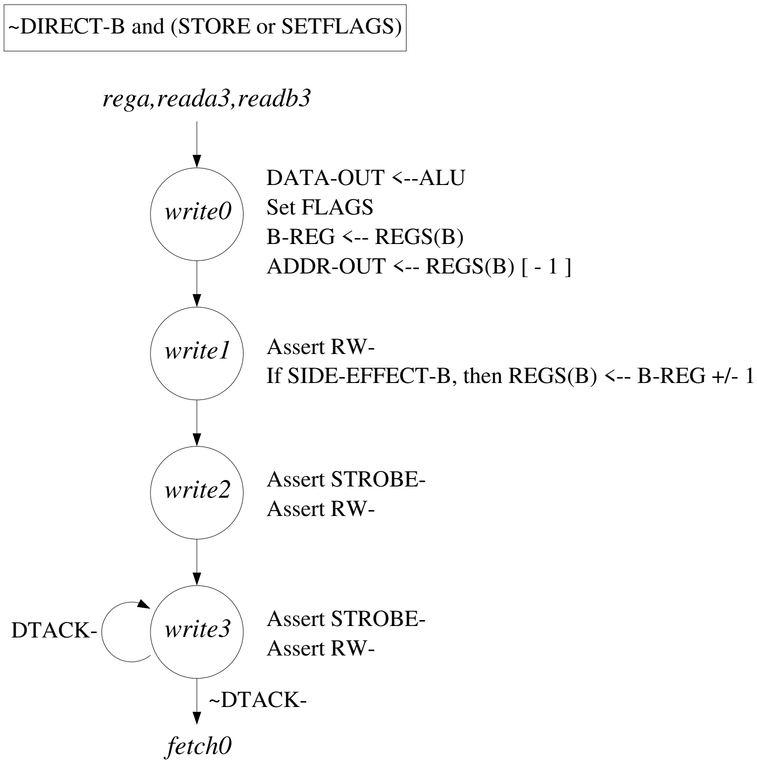


Figure 10.7: Major Control State Diagram 4

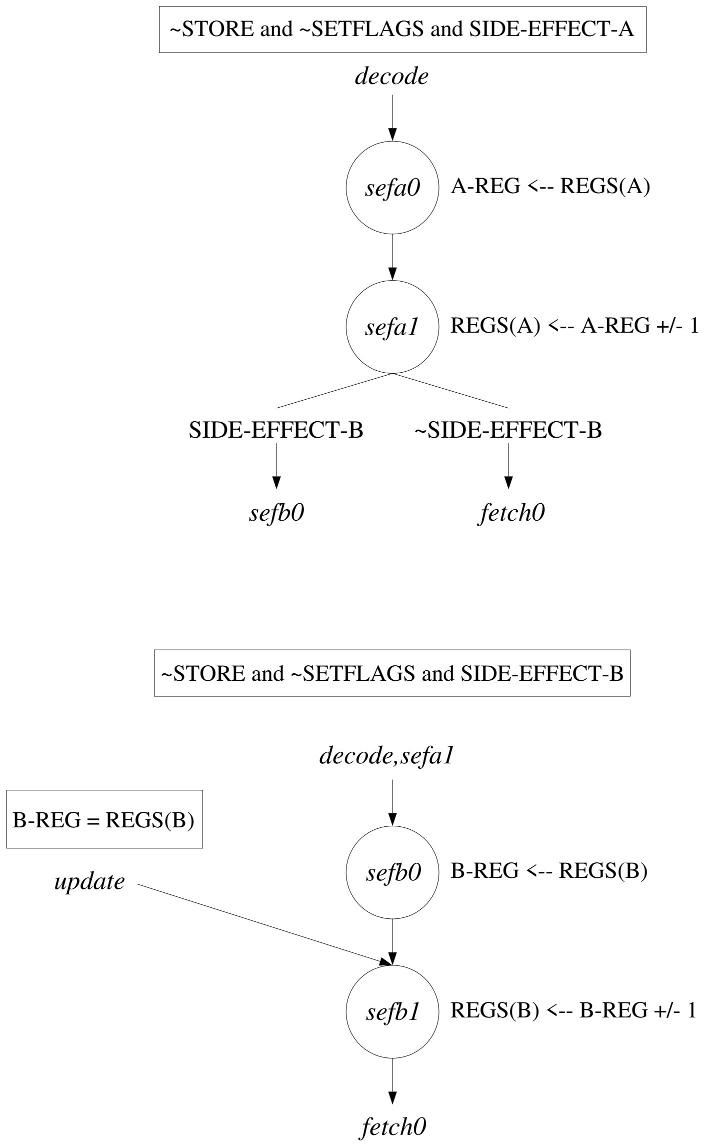


Figure 10.8: Major Control State Diagram 5

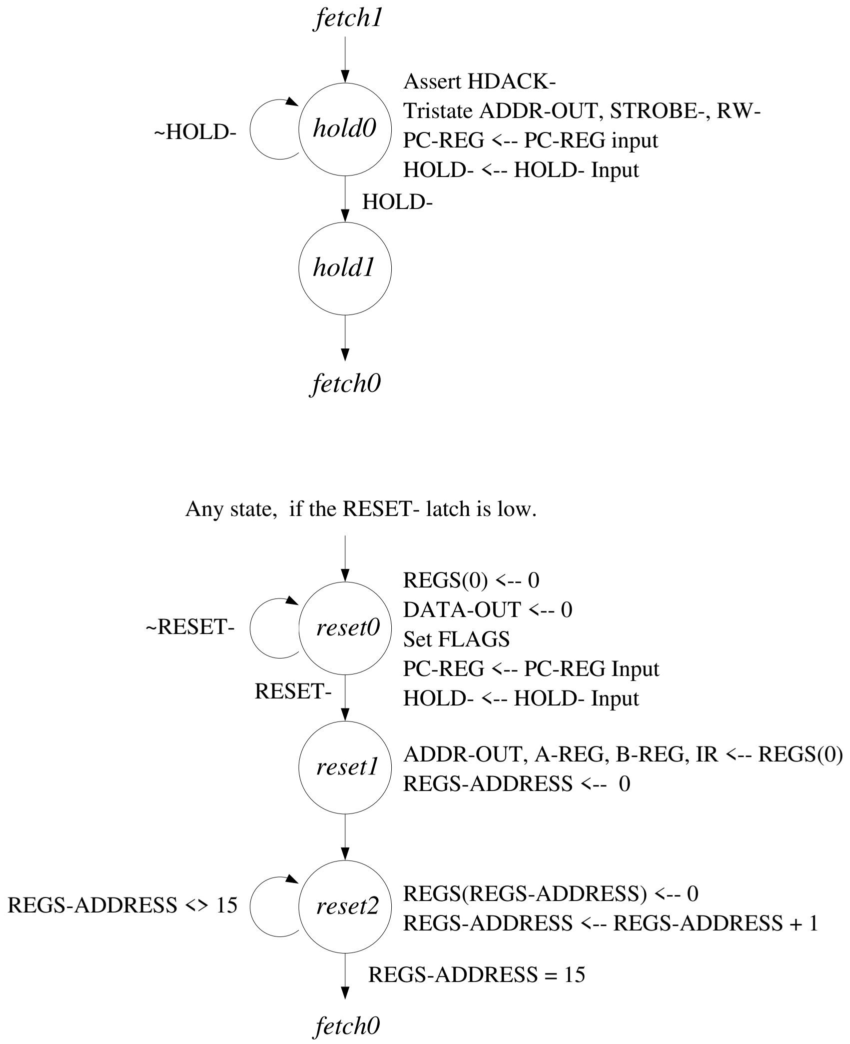


Figure 10.9: Major Control State Diagram 6

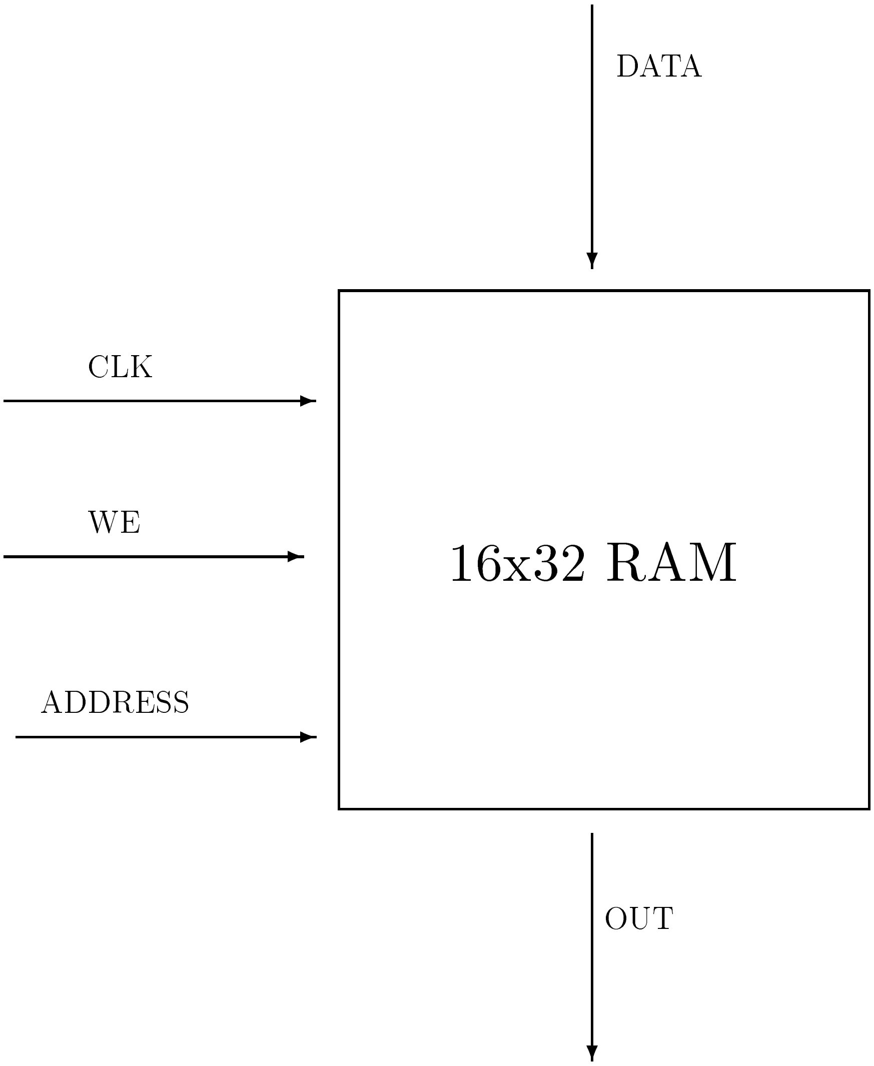


Figure 10.10: FM9001 Register File

address `ADDRESS`. The register file can only be read or written each clock cycle (not both).

In order to save space on the chip, we chose to use a level-sensitive RAM to implement the register file. We have surrounded this RAM with several 32-bit latches, in order to make our edge-triggered model of state faithful to the "snap-shot" semantics embodied by `DUAL-EVAL`, as described in Section 5.1. Figure 10.11 is a more detailed schematic for the register file that displays the various latches. Note that the level-sensitive RAM takes a clock cycle in order to store input data, which means that we need to take special care when attempting to read data from a register that is currently being written. Therefore we use a comparator and a multiplexor to allow accurate "reading" of the data stored during the prior clock cycle.

Let us take a look at the Boolean behavioral specifications of the read and write operations for the register file. The read operation follows the description above: if the address agrees with the previous clock cycle's address that was latched in at that time, and we are currently performing a write operation, then we need to pass the data straight through to the output; otherwise, we simply get the value from the level-sensitive RAM.

Definition.
```
(read-regs address regs)
  =
(let ((regfile      (car regs))
      (last-we      (cadr regs))
      (last-data    (caddr regs))
      (last-address (cadddr regs)))
  (if (and last-we (v-iff address last-address))
      last-data
    (read-mem address regfile)))
```

The behavioral specification of the write operation returns a new state, where the state contains four components: the new RAM, the latched-in write-enable value, and the appropriate data and address values.

Definition.
```
(write-regs we address regs data)
 =
(let ((regfile      (car regs))
      (last-we      (cadr regs))
      (last-data    (caddr regs))
      (last-address (cadddr regs)))
  (list (if last-we
            (write-mem last-address regfile last-data)
          regfile)
        we
```

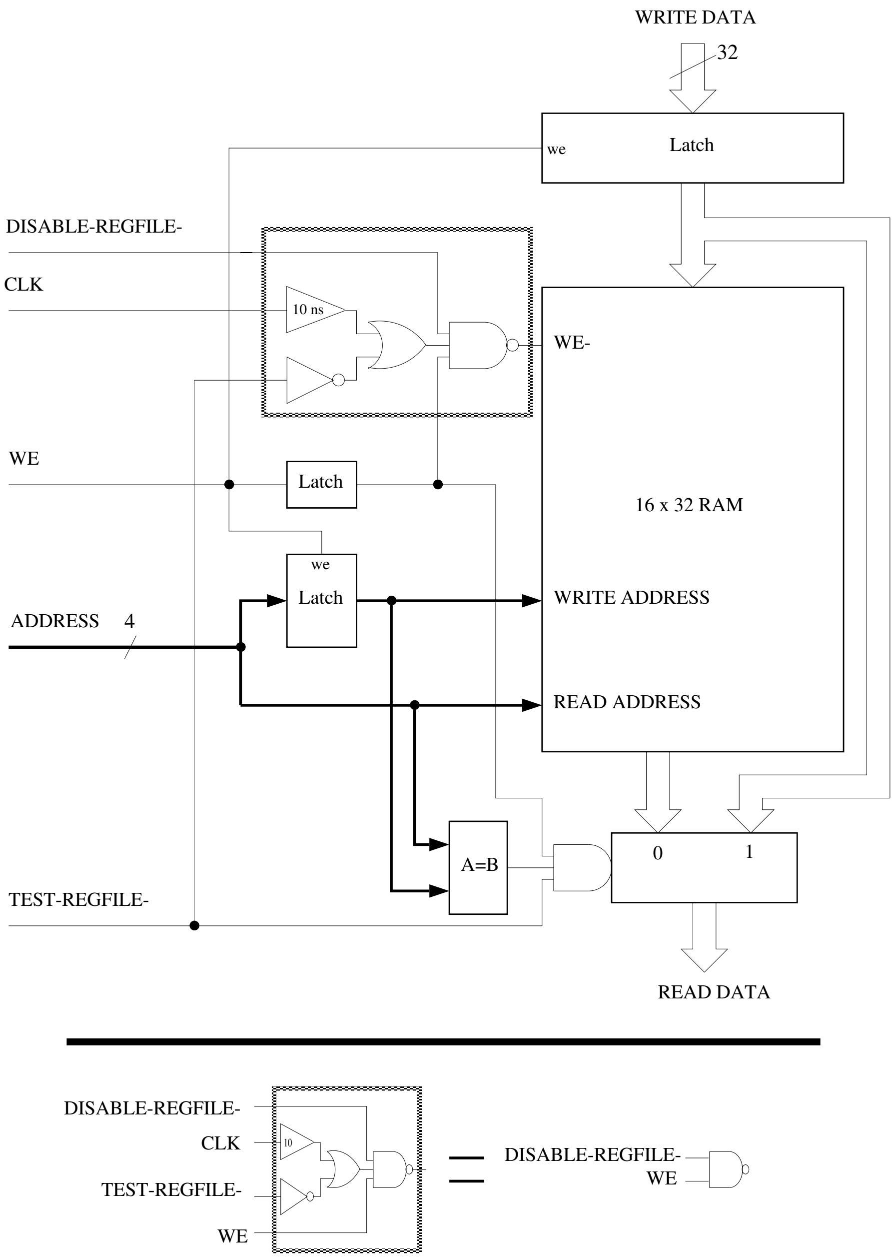


Figure 10.11: A Detailed View of the FM9001 Register File. Serial scan interconnections are not shown. The `DUAL-EVAL` model of the boxed circuit is a special NAND gate primitive, `RAM-ENABLE-CIRCUIT`.

```
        (if we data last-data)
        (if we address last-address)))
```

Our implementation of the register file is produced using the `MODULE-GENERATOR` macro (see Section 2.6). If we were to build what we want in the most straightforward way, the clock would be an input to a delay gate, yet our top-level predicate does not allow clock signals to be used in that way. Therefore, we provide an artificial primitive called `RAM-ENABLE-CIRCUIT` that provides the desired clocking for the level-sensitive RAM. However, our translator to LSI Logic's Network Description Language (NDL) includes special code to generate an implementation in actual gates rather than a reference to our artificial primitive. This primitive, preceded by a comment explaining its implementation in gates, may be found (of course) in the primitives file, "primp-database.lisp" (see Section 2.9).

See Section 14.59 for the event file "regfile.events".

10.5 FLAGS

This file defines the hardware for updating the flags and proving the usual value and state theorems. We use a scan latch primitive `FD1SLP` for each flag for easier testing.

See Section 14.60 for the event file "flags.events".

10.6 EXTEND-IMMEDIATE

The `EXTEND-IMMEDIATE` module sign-extends a 9-bit input vector to a 32-bit vector. This module is required for the case that the machine instruction indicates that the A operand is to be used as immediate data; see Section 9.1.

See Section 14.61 for the event file "extend-immediate.events".

10.7 PAD-VECTORS

Signal strengths inside the chip are smaller than the strengths on the external input and output lines. The various I/O pad devices provide an appropriate interface.

In this file we define n-bit module generators for four types of I/O pads: `TTL-INPUT`, `TTL-OUTPUT`, `TTL-TRI-STATE-OUTPUT`, and `TTL-BIDIRECTIONAL`. For example, a group of three `TTL-INPUT` pads is defined as follows. We use the function `LISP-NETLIST` (see Section 5.9) to make the result more readable.

```
(lisp-netlist (ttl-input-pads* 3))
 =
'(TTL-INPUT-PADS_3 (PI IN_0 IN_1 IN_2)
                   (PO_2 B-OUT_0 B-OUT_1 B-OUT_2)
```

```
((G_0 (OUT_0 PO_0) TTL-INPUT (IN_0 PI))
 (B_0 (B-OUT_0) B-BUF (OUT_0))
 (G_1 (OUT_1 PO_1) TTL-INPUT (IN_1 PO_0))
 (B_1 (B-OUT_1) B-BUF (OUT_1))
 (G_2 (OUT_2 PO_2) TTL-INPUT (IN_2 PO_1))
 (B_2 (B-OUT_2) B-BUF (OUT_2)))
NIL)
```

Actually, the `B-BUF` buffers are an artifact of LSI Logic's software. A module must contain at least one internal gate on the inside; it may not consist entirely of pads. Buffers are placed on every pad data output, instead of on just one pad output, so that the timing for all input pads is similar. The outputs `PO_0`, `PO_1`, and `PO_2` are parametric pad outputs that are used for post-manufacture testing of the input pads.

See Section 14.62 for the event file "pad-vectors.events".

10.8 FM9001-HARDWARE

We are now almost ready to build the chip system (chip with memory) using the modules that we have been defining. First, however, we specify the behavior of the chip system under normal operating conditions. Thus, we assume that the test lines `TE`, `DISABLE-REGFILE-` and `TEST-REGFILE-` are inactive. The theorem that makes these assumptions and equates the chip system's operation with its behavioral specification is found in the next section, Section 10.9. Thus (ironically), in spite of its name, this is the only file in the present chapter that does not actually define hardware.

The low-level (four-valued logic) specification of the chip's operation under non-test conditions is given by the following function.

Definition.
```
(run-fm9001 state oracle n)
 =
(if (zerop n)
    state
  (run-fm9001 (fm9001-next-state state (car oracle))
              (cdr oracle)
              (sub1 n)))
```

The following lemma has analogues in many interpreter proofs.

Theorem. `RUN-FM9001-PLUS`

```
(equal (run-fm9001 state oracle (plus n m))
       (run-fm9001 (run-fm9001 state oracle n)
                   (nthcdr n oracle)
                   m))
```

This file also has a number of lemmas stating basic structural properties of the chip state.

See Section 14.63 for the event file "fm9001-hardware.events".

10.9 CHIP

This file glues together the hardware modules we have created so far and creates the final chip system (chip with memory), in the following three stages.

- `CHIP-MODULE`: The entire internal logic of the chip, but without the I/O pads.
- `CHIP`: The entire chip, including the I/O pads.
- `CHIP-SYSTEM`: The entire chip, bus, and memory system.

Documentation appears in the file "chip.events". For all three of these modules we prove lemmas stating the equivalence of the `DUAL-EVAL` semantics of the modules with low-level structural-style specifications.

There are a number of unusual (but important) lemmas that are used as hints. For example, the following lemma tells the theorem prover how to write a list as an explicit call of `APPEND` on appropriate sublists.

Theorem. `EQUAL-LENGTH-40-AS-COLLECTED-NTH+SUBRANGE`

```
(implies
 (and (equal (length l) 40)
      (properp l))
 (equal l (append (list-as-collected-nth l 16 0)
                  (subrange l 16 39))))
```

The file concludes with the following theorem, which is the main theorem in the file. It says that under certain hypotheses (to be explained below), the result of running `DUAL-EVAL` for `N` steps from an appropriate initial `STATE`, using an appropriate oracle `INPUTS`, agrees with our low-level (four-valued) specification function `RUN-FM9001`.

Theorem `CHIP-SYSTEM=RUN-FM9001`

```
(implies
```

```
  (and (chip-system& netlist)
       (fm9001-state-structure state)
       (chip-system-invariant state)
       (chip-system-operating-inputs-p inputs n)
       (equal fm9001-inputs (map-up-inputs inputs)))
  (equal (simulate-dual-eval-2 'chip-system inputs state netlist n)
         (run-fm9001 state fm9001-inputs n)))
```

This theorem states that the netlist level implements the four-valued level, as illustrated in Figure 1.1. Because of this theorem, we can turn our attention to proving the equivalence of the low-level specification function RUN-FM9001, which does not mention any netlists but is purely "logical," with a higher-level specification function. Thus, we have proven that the structural definition of the FM9001 does indeed implement a more abstract, but still low-level specification function.

We now explain the hypotheses of the theorem above. The first ones are simple. (CHIP-SYSTEM& NETLIST) says that NETLIST contains our definition of the FM9001 netlist, including the memory and interconnection bus. The conjunction of the predicates (FM9001-STATE-STRUCTURE STATE) (cf. Section 10.8) and (CHIP-SYSTEM-INVARIANT STATE) says that the state has the right shape.

This file also defines certain properties of the CHIP-SYSTEM's state and the input oracle, which is just a list of input vectors for the CHIP-SYSTEM; see the discussion of the INPUTS argument to SIMULATE-DUAL-EVAL-2 near the end of Section 5.1. We assume that the test lines are disabled and the register file is enabled by assuming that CHIP-SYSTEM-INPUT-INVARIANT holds.

Definition.

```
(chip-system-input-invariant inputs)
 =
(let ((clk              (car inputs))
      (ti               (cadr inputs))
      (te               (caddr inputs))
      (reset-           (cadddr inputs))
      (hold-            (caddddr inputs))
      (disable-regfile- (cadddddr inputs))
      (test-regfile-    (caddddddr inputs))
      (pc-reg-in        (cdddddddr inputs)))
  (and (equal te f)
       (equal disable-regfile- t)
       (equal test-regfile- t)
       (properp pc-reg-in)
       (equal (length pc-reg-in) 4)))
```

Now we are ready to explain the next hypothesis (`CHIP-SYSTEM-OPERATING--INPUTS-P INPUTS N`). It says that each input vector in the oracle satisfies the predicate whose definition is displayed just above.

Finally, we need the notion of "mapping up" an oracle. The low-level oracle supplies an input vector for the `CHIP-SYSTEM` at each clock tick, and so does the higher-level oracle. The difference is that the higher-level oracle does not contain any test inputs. This makes sense, since our theorem assumes (with `CHIP-SYSTEM-INPUT-INVARIANT`) that the chip system is operating in a non-test mode, and hence the test lines are not appropriate for the higher-level oracle. Thus, in effect, we have proven that the test logic does not interfere with the "normal" operation of the chip.

See Section 14.64 for the event file "chip.events".

Chapter 11

CORRECTNESS OF THE FM9001

The most important theorems proved about the FM9001 are contained in the following files:

- "proofs.events" — the principal result that the netlist implements the high level
- "approx.events" — the monotonicity of the `DUAL-EVAL` HDL
- "final-reset.events" — that the processor can be reset and a more refined version of lemmas from "proofs.events"
- "well-formed-fm9001.events" — that the netlist is well-formed
- "alu-interpretation.events" — correctness of the ALU with respect to natural number and integer operations
- "flag-interpretation.events" — correctness of arithmetic flag results
- "more-alu-interpretation.events" — refinements of "alu-interpretation.events"

These files and several others are discussed in the next three chapters.

11.1 EXPAND-FM9001

Recall that `FM9001-NEXT-STATE` is the single-step function for the FM9001 low-level (four-valued level) interpreter, `RUN-FM9001`.

In order to prove that the control logic in the FM9001 correctly executes machine instructions, we prove rewrite rules that, in essence, step the low-level machine. That is, we prove rewrite rules to allow the prover to symbolically execute the low-level machine from any major control state to the major control states that follow. Consider, for example, the first of these forms:

```
(STEP-FM9001 FETCH0 :SUFFIX 0)
```

which submits the large event FETCH0$STEP0 to the prover.

We submit a number of such symbolic single-step simulations to the prover. Then we prove, by induction, for each place in the state diagrams where a one-cycle loop exists, that we can wait in this loop for an indeterminate amount of time. After proving all the single-step lemmas for the major control states, we combine them into multi-step simulations. We know of no other microprocessor verification where it has been proven that arbitrary delays by the memory are acceptable.

See Section 14.65 for the event file "expand-fm9001.events".

11.2 PROOFS

The first part of this file establishes, in lemma FM9001-INTERPRETER-CORRECT, that the four-valued level implements the Boolean user-level description of the FM9001. This proof makes heavy use of the "chunk" simulations performed by the lemmas in the file "expand-fm9001.events" (see Section 11.1).

```
Theorem.  FM9001-INTERPRETER-CORRECT

(let ((clock-cycles (total-microcycles state inputs instructions)))
  (implies
   (and (fm9001-state-structure state)
        (macrocycle-invariant state pc)
        (operating-inputs-p inputs clock-cycles))
   (equal (map-up (run-fm9001 state inputs clock-cycles))
          (FM9001-interpreter (map-up state) pc instructions))))
```

We now paraphrase the preceding theorem. The function TOTAL-MICROCYCLES, given the current state of the machine, the machine inputs, and the number of instructions to be executed, computes how many low-level steps (clock cycles) are required by the four-valued level interpreter, RUN-FM9001, to complete the same number of user-level instructions.

The predicate FM9001-STATE-STRUCTURE insures that the STATE is well formed. The MACROCYCLE-INVARIANT predicate is an invariant about the state of the machine at the beginning of the instruction execution cycle: all of the internal registers are Boolean and properly sized; the machine is in state FETCH1 (ready to execute an instruction); the `ADDR-OUT` register contains the program counter; there are no pending writes in the register file; `HOLD-` and `RESET-` are not asserted; the `PC-REG` is equal to (and remains equal to) the specified `PC`; and the memory is well-formed and in a quiescent state. This is an important invariant; one of the more important proofs that follows states that if we begin in this state, and run for `(MICROCYCLES STATE)`, then we will reach a state recognized by this invariant. This is the basis for the inductive proof that the four-valued level machine implements the behavioral specification.

The predicate `OPERATING-INPUTS-P` recognizes those four-valued level input streams that do not reset the machine and that do not include hold requests. The function `MAP-UP` converts a four-valued level state to a user-level state.

The conclusion states that running the four-valued interpreter `RUN-FM9001` the proper number of clock cycles produces the same user-level state as the user-level specification `FM9001-INTERPRETER`.

The following theorem ties the above theorem together with the last theorem in the file "chip.events" to establish that the netlist level specification of the FM9001, when interpreted by the `DUAL-EVAL` simulator, implements the user-level specification.

```
Theorem.  CHIP-SYSTEM=FM9001-INTERPRETER

(let ((rtl-inputs (map-up-inputs inputs)))
  (let ((clock-cycles
         (total-microcycles state rtl-inputs instructions)))
    (implies
     (and (chip-system& netlist)
          (fm9001-state-structure state)
          (macrocycle-invariant state pc)
          (chip-system-operating-inputs-p inputs clock-cycles)
          (operating-inputs-p rtl-inputs clock-cycles))
     (equal
      (map-up (simulate-dual-eval-2
               'chip-system inputs state netlist clock-cycles))
      (fm9001-interpreter (map-up state) pc instructions)))))
```

The theorem above is the first of a number of similar statements that the FM9001 user-level specification is indeed implemented by the netlist design. Looking back at Figure 1.1, this theorem states that “the diagram commutes,” i.e., speaking in informal programmer’s jargon, if one starts with a low-level state, maps it up to a high-level state, and runs the user-level FM9001 specification on that state thereby obtaining a final state, one may also obtain the same final state by instead simulating the FM9001 netlist using `DUAL-EVAL` on the initial low-level state and then mapping that result up. The preceding sentence ignores the issues of external memory responses. Let us now review the hypotheses and conclusions of `CHIP-SYSTEM=FM9001-INTERPRETER` less informally. Let `RTL-INPUTS` be the external inputs to the FM9001 after stripping out the test inputs. Further, let `CLOCK-CYCLES` be the number of clock cycles required by the microarchitecture to complete the number of instructions specified by the natural number `INSTRUCTIONS`. The first hypothesis `(CHIP-SYSTEM& NETLIST)` requires that `NETLIST` contain the netlist for the entire microprocessor and memory system. The second hypothesis requires that the low-level state be well formed; we later prove that the reset process will produce such a state. The third hypothesis, `(MACROCYCLE-INVARIANT STATE PC)`, requires that all of the internal registers be Boolean and properly sized; the machine be in state `FETCH1` (ready to execute an instruction); the `ADDR-OUT` register contain the program counter; that there be no pending writes in the register file; that `HOLD-` and `RESET-` are not asserted; that `PC-REG` be equal to (and remain equal to) the specified `PC`; and that the memory be well-formed and in a quiescent state. The fourth hypothesis, `(CHIP-SYSTEM-OPERATING-INPUTS-P INPUTS CLOCK-CYCLES)`, stipulates that test inputs to the FM9001 are disabled. The last hypothesis, `(OPERATING--INPUTS-P RTL-INPUTS CLOCK-CYCLES)`, requires that the external reset and hold signals remain disabled.

In the conclusion, the function `MAP-UP` converts a well-formed FM9001 netlist-level state to a user-level state. `SIMUALTE-DUAL-EVAL-2` repeatedly applies `DUAL--EVAL` to simulate the netlist. `FM9001-INTERPRETER` is the behavioral level specification for the FM9001; this is the arrow named "Boolean Interpretation" in Figure 1.1. We later prove that the arrows named "Integer Interpretation" and "Natural Number Interpretation" are equivalent to the "Boolean Interpretation" arrow (see Chapter 13).

Here is an informal statement of the chief result proved about the FM9001. Let h be the hardware netlist that we have constructed for the FM9001. We have proved that for each FM9001 user-visible state, s, and for each positive integer, n, there exists a positive integer c such that the result of running the user-model of the FM9001 (the function `FM9001`) n steps can instead be obtained by simulating h and s under the `DUAL-EVAL` semantics for c steps. In precisely stating the theorem, we arrange to supply the `DUAL-EVAL` semantics with a transform (obtained with `MAP-DOWN`) of s, and afterwards we do a reverse transformation (with `MAP-UP`) to obtain the final user-visible state. Futhermore, in making the statement precise we stipulate that the external stimuli, i, to the chip do not enable a hold, a reset, or a test input for the c clock cycles. The fact that n and c are different reflects the fact that a single FM9001 instruction takes several clock cycles to implement at the `DUAL-EVAL` netlist level. The precise statement of this correctness result is:

```
Theorem.  FM9001=CHIP-SYSTEM-SUMMARY

(let ((h (chip-system$netlist))
      (c (total-microcycles (map-down s) (map-up-inputs i) n)))
```

```
(implies
 (and (fm9001-statep s)
      (no-holds-reset-or-test i c))
 (equal
  (fm9001 s n)
  (map-up
   (simulate-dual-eval-2 'chip-system i (map-down s) h c)))))
```

This theorem is the final event of the file "proofs.events". See the file "final-reset.events" for a set of lemmas that prove that such a state can be reached as a result of the reset process.

11.3 APPROX

An important property of the FM9001 is that a specified *reset sequence* of input vectors puts the FM9001 in an appropriate specified initial state. The proof of this property can be done by direct simulation if one views the `(X)` value as truly representing the 'unknown' value, because if one can show that the reset sequence produces the desired effect starting with an initial state of all `(X)` values, then it ought to produce the desired effect starting with *any* initial state. The point of the present event file is to show that in fact, it really does suffice to consider an initial state consisting of all `(X)` values in order to make the desired conclusion about an arbitrary initial state. A more complete description of this event file may be found in Kaufmann's "A Hardware Reset Lemma and Its Proof"[18].

In order to motivate what follows, let us begin by considering what our four-valued logic functions return when some of their inputs are the "unknown" value, `(X)`. In elementary mathematics, we say that a function $f(x)$ is monotonic provided that

$$x \leq y \rightarrow f(x) \leq f(y)$$

If we think of the value `(X)` as being "strictly less-well defined" than the values `T` or `F`, then we can check that `F-AND`'s output is weakly more well-defined on input that is more weakly well-defined. In other words, `F-AND` is monotonic in both its arguments.

```
(f-and (X) T) = (X)

(f-and T T) =  T

(f-and (X) F) =  F

(f-and T F) = F
```

The preceding facts can be proved by mere execution, using `R-LOOP`.

Here is the final, cruical event in the file, followed by some comments.

```
Theorem.  XS-SUFFICE-FOR-RESET-LEMMA
(let ((final-1 (nth n (simulate fn inputs state-1 netlist)))
      (final-2 (nth n (simulate fn inputs state-2 netlist))))
  (implies (and (lessp n (length inputs))
                (s-approx state-1 state-2)
                (v-knownp (car final-1))
                (s-knownp (cadr final-1))
                (ok-netlistp 0 fn netlist '(mem-32x32)))
           (equal final-1 final-2)))
```

This lemma may be read as follows; we comment on the text of the lemma in order of appearance.

Consider two simulations whose final "snapshots" (see Section 5.1) are denoted `FINAL-1` and `FINAL-2` that start (respectively) from states `STATE-1` and `STATE-2`.

```
(let ((final-1 (nth n (simulate fn inputs state-1 netlist)))
      (final-2 (nth n (simulate fn inputs state-2 netlist))))
  ...)
```

Assume that we have a sequence `INPUTS` containing more than `N` input vectors.

```
(lessp n (length inputs))
```

Assume that `STATE-1` *approximates* `STATE-2`, roughly speaking in the sense that `STATE-2` may differ from `STATE-1` only by replacing some `(X)` values in `STATE-1` by any other values.

```
(s-approx state-1 state-2)
```

The result `FINAL-1` of the first simulation is actually a pair consisting of the final output vector together with the final state. With that in mind, we can view the next two assumptions as stating that the final output vector contains no `(X)` value and the final state also contains no `(X)` value.

```
(v-knownp (car final-1))
(s-knownp (cadr final-1))
```

Finally, assume that the netlist is syntactically well-formed, in the sense that all "leaf" modules are primitives, none of which is the memory primitive.

```
(ok-netlistp 0 fn netlist '(mem-32x32))
```

Then the two final output vectors are the same and the two final states are the same.

```
(equal final-1 final-2)
```

Let us sketch the strategy we use for proving the lemma above. The main idea is that the semantics given by `DUAL-EVAL` (cf. Section 5.1) is *monotone*. Consider the relation `B-APPROX` that holds between `(X)` and the other four-valued logic values, and also holds between any value and itself. More formally, we may say that a value is *known* if and only if it is either `T`, `F`, or `(Z)`. In particular, `(X)` is not 'known'. Then we formalize the notion of approximation for logic values as follows.

```
Definition.
(b-knownp x)
 =
(or (equal x t)
    (equal x f)
    (equal x (z))))

Definition.
(b-approx a1 a2)
 =
(if (b-knownp a1)
    (equal a1 a2)
  t))
```

This notion of approximation extends naturally to vectors (see the definition of `V-APPROX`) and state (see `S-APPROX`). Then the lemma `DUAL-EVAL-MONOTONE` says that `DUAL-EVAL` with flags 0 and 2 preserves the approximation ordering, as follows.

```
Lemma. DUAL-EVAL-MONOTONE  ;; slightly massaged
(implies (and (ok-netlistp flag fn netlist '(mem-32x32))
              (v-approx a1 a2)
              (s-approx s1 s2))
         (and (v-approx (dual-eval 0 fn a1 s1 netlist)
                        (dual-eval 0 fn a2 s2 netlist))
              (s-approx (dual-eval 2 fn a1 s1 netlist)
                        (dual-eval 2 fn a2 s2 netlist))))
```

It follows fairly easily from this lemma that simulation, i.e., the repeated application of `DUAL-EVAL` to a sequence of input vectors, is also monotone in an appropriate sense. Since "known" values can only approximate themselves, the lemma `XS-SUFFICE-FOR-RESET-LEMMA` shown at the start of this section follows readily.

Finally, we discuss briefly the idea of the proof of `DUAL-EVAL-MONOTONE`. The plan is to start by proving that the basic four-valued logic operators are monotone, then prove that the hardware primitives are monotone, and finally do the inductive proof that `DUAL-EVAL` is monotone. The file "monotonicity-macros.lisp" is quite helpful for the first two of these three steps, as we now show.

Consider the form `(MONOTONICITY-LEMMAS (F-BUF F-AND))` from the present file, "approx.events". It creates the following lemmas, which say that the logic functions `F-BUF` and `F-AND` are monotone.

```
(PROVE-LEMMA F-BUF-MONOTONE (REWRITE)
  (IMPLIES (B-APPROX A1 A2)
           (B-APPROX (F-BUF A1) (F-BUF A2))))

(PROVE-LEMMA F-AND-MONOTONE (REWRITE)
  (IMPLIES (AND (B-APPROX A1 A2) (B-APPROX B1 B2))
           (B-APPROX (F-AND A1 B1) (F-AND A2 B2))))
```

Similar calls of the macro `MONOTONICITY-LEMMAS` create similar lemmas for the rest of the primitives, sometimes with hints. Consider for example the following form.

```
(monotonicity-lemmas (f-and3 f-and4) ((disable f-and b-approx)))
```

This creates lemmas for `F-AND3` and `F-AND4` analogous to the ones displayed above, but with the indicated `DISABLE` hint appended.

The next task is to prove monotonicity lemmas for the primitive state-holding hardware modules, except the memory. The macro constructed for this purpose is called `PROVE-PRIMITIVE-MONOTONICITY`. In this case, all primitive modules except the register file and the memory are handled by a single application of this macro. However, for illustrative purposes let us look at the events created by a call of this macro on just two primitives: `(PROVE-PRIMITIVE-MONOTONICITY (B-AND FD1))`. A quick perusal will show that this macro creates some rather sophisticated hints; these help the theorem prover perform the required proofs. The important lemmas below are `B-AND-MONOTONE` and `FD1-MONOTONE`, which say respectively that that a primitive two-input AND gate and a D flip-flop respect the approximation order. The notion `MONOTONICITY-PROPERTY` simply says that the approximation order is preserved, as in the statement of the lemma `DUAL-EVAL-MONOTONE` shown above.

```
(PROVE-LEMMA DUAL-EVAL-B-AND-VALUE (REWRITE)
  (EQUAL (DUAL-EVAL 0 'B-AND ARGS STATE NETLIST)
         (LET ((A (CAR ARGS)) (B (CAR (CDR ARGS))))
              (CONS (F-AND A B) 'NIL)))
  ((ENABLE DUAL-EVAL DUAL-APPLY-VALUE)))

(PROVE-LEMMA DUAL-EVAL-B-AND-STATE (REWRITE)
  (EQUAL (DUAL-EVAL 2 'B-AND ARGS STATE NETLIST) 0)
  ((ENABLE DUAL-EVAL DUAL-APPLY-STATE)))

(PROVE-LEMMA B-AND-MONOTONE (REWRITE)
  (AND (MONOTONICITY-PROPERTY 0 'B-AND NETLIST A1 A2 S1 S2)
       (MONOTONICITY-PROPERTY 2 'B-AND NETLIST A1 A2 S1 S2))
  ((DISABLE-THEORY T)
   (ENABLE-THEORY GROUND-ZERO MONOTONICITY-LEMMAS)
   (ENABLE *1*B-APPROX *1*V-APPROX *1*S-APPROX V-APPROX
           MONOTONICITY-PROPERTY-OPENER-0
           MONOTONICITY-PROPERTY-OPENER-2 DUAL-EVAL-B-AND-VALUE
           DUAL-EVAL-B-AND-STATE S-APPROX-IMPLIES-B-APPROX
           FOURP-IMPLIES-S-APPROX-IS-B-APPROX FOURP-F-BUF
           FOURP-F-IF)
   (EXPAND (V-APPROX A1 A2) (V-APPROX (CDR A1) (CDR A2)))))

(PROVE-LEMMA DUAL-EVAL-FD1-VALUE (REWRITE)
  (EQUAL (DUAL-EVAL 0 'FD1 ARGS STATE NETLIST)
         (LET ((D (CAR ARGS)) (CP (CAR (CDR ARGS))))
              (CONS (F-BUF STATE) (CONS (F-NOT STATE) 'NIL))))
  ((ENABLE DUAL-EVAL DUAL-APPLY-VALUE)))

(PROVE-LEMMA DUAL-EVAL-FD1-STATE (REWRITE)
  (EQUAL (DUAL-EVAL 2 'FD1 ARGS STATE NETLIST)
         (LET ((D (CAR ARGS)) (CP (CAR (CDR ARGS))))
              (F-BUF D)))
  ((ENABLE DUAL-EVAL DUAL-APPLY-STATE)))

(PROVE-LEMMA FD1-MONOTONE (REWRITE)
  (AND (MONOTONICITY-PROPERTY 0 'FD1 NETLIST A1 A2 S1 S2)
       (MONOTONICITY-PROPERTY 2 'FD1 NETLIST A1 A2 S1 S2))
  ((DISABLE-THEORY T)
   (ENABLE-THEORY GROUND-ZERO MONOTONICITY-LEMMAS)
   (ENABLE *1*B-APPROX *1*V-APPROX *1*S-APPROX V-APPROX
           MONOTONICITY-PROPERTY-OPENER-0
           MONOTONICITY-PROPERTY-OPENER-2 DUAL-EVAL-FD1-VALUE
           DUAL-EVAL-FD1-STATE S-APPROX-IMPLIES-B-APPROX
           FOURP-IMPLIES-S-APPROX-IS-B-APPROX FOURP-F-BUF
           FOURP-F-IF)
   (EXPAND (V-APPROX A1 A2) (V-APPROX (CDR A1) (CDR A2)))))
```

The macro `DISABLE-ALL` disables all of its arguments. The macro `REVERT-STATE` is intended to make the set of enabled rules be the same as those in existence just after the indicated event. In particular, the form `(REVERT-STATE BVP-REV1)` reverts to the state just after the lemma `BVP-REV1` in "approx.events".

See Section 14.67 for the event file "approx.events".

Chapter 12

ADDITIONAL PROPERTIES OF THE CHIP

In this chapter we conclude our proofs about the FM9001 netlist by showing that the chip's reset sequence has the desired effect and that the netlist is well-formed.

12.1 FINAL-RESET

The theorem `XS-SUFFICE-FOR-RESET-CHIP-LEMMA-INSTANCE` states that if the FM9001 is run starting in a completely unknown state, then after nineteen clock cycles after the reset input is disabled, the internal state of the FM9001 has been set to known values and is ready to execute user instructions. This reset process does not depend upon any internal state value, but does require that the reset line be asserted at the beginning of the reset sequence. One can check with simulation on any specific initial state that after the reset sequence that the FM9001 internal state is equal to `(FINAL-STATE)`.

The last part of this file contains the most general statements concerning the correctness of the FM9001 design. The final lemma `CHIP-SYSTEM=FM9001--INTERPRETER$AFTER-RESET` is the similar to the lemma named `CHIP-SYSTEM=-FM9001-INTERPRETER` in the file "proofs.events", except that it is specialized to the case where the initial state is made up of the chip state after reset (i.e., `(INITIALIZED-MACHINE-STATE)`) together with an appropriate machine memory. Thus, the hypothesis `(FM9001-STATE-STRUCTURE STATE)` has been omitted and the rather elaborate hypothesis `(MACROCYCLE-INVARIANT STATE)` has been replaced by the much weaker hypothesis `(MEMORY-OKP 32 32 MEMORY)`. We can do this because `(INITIALIZED-MACHINE-STATE)` satisfies the processor state portions of these two hypotheses.

See Section 14.68 for the event file "final-reset.events".

12.2 WELL-FORMED-FM9001

The LSI logic tools require that certain syntactic conditions be met on the netlist, as well as the conditions we impose on our HDL; see Section 5.4 for a description of `TOP-LEVEL-PREDICATE`. The three `PROVE-LEMMA` events in this file show that all of these requirements are met.

See Section 14.69 for the event file "well-formed-fm9001.events".

Chapter 13

ALU INTERPRETATION LEMMAS

In this chapter we demonstrate the correctness of the results produced by the ALU and prove the correctness of the values stored in the flag registers. Looking back at Figure 1.1, we show that the "Integer Interpretation" and "Natural Number Interpretation" are equivalent to the the "Boolean Interpretation." We recommend that the reader, when interested in the ALU interpretations, skip reading the file "alu-interpretation.events" in favor of the file "more-alu-interpretation.events".

13.1 MATH-ENABLE

This file enables appropriate arithmetic lemmas after "winding back" to an appropriate state. See Section 2.5 for more information.

See Section 14.70 for the file "math-enable.events".

13.2 ALU-INTERPRETATION

The two main theorems in this file state that the ALU behaves as expected when its inputs are viewed as natural numbers or as integers (respectively). Let us take a look at the natural number version.

```
Theorem.  V-ALU-CORRECT-NAT
(implies (bv2p a b)
         (equal (v-alu c a b op)
                (v-alu-nat c a b op)))
```

The hypothesis `(BV2P A B)` says that `A` and `B` are bit vectors of the same length. Henceforth, let us imagine particular values of `C`, `A`, `B`, and `OP`, and assume that `(BV2P A B)` is true. Furthermore, in order to begin to understand the conclusion of the theorem above, let us see what it says in the special case that `OP` is the value `#v0010`, which happens to represent the add-with-carry operation.

In that case, and under our assumption of `(BV2P A B)`, the theorem immediately reduces to the following term when we open up the definitions of `V-ALU` and `V-ALU-NAT`, where again we take `OP` to equal `#v0010`.

```
(equal (cvzbv-v-adder c a b)
       (v-alu-nat-adder c a b))
```

The first argument of the `EQUAL` term above represents the value returned by the ALU when doing an add-with-carry operation, according to its low-level specification. The second argument of this term returns the value that the natural-number specification says the ALU ought to return.

If we open up the definitions of `CVZBV-V-ADDER` and `V-ALU-NAT-ADDER`, then the equality displayed above reduces to the following term.

```
(equal (cvzbv (v-adder-carry-out c a b)
              (v-adder-overflowp c a b)
              (v-adder-output    c a b))
       (cvzbv (v-alu-nat-adder-carry-out c a b)
              (v-adder-overflowp         c a b)
              (v-alu-nat-adder-output    c a b)))
```

It follows easily from the equality above and the definition of `CVZBV` that the following two equalities hold.

```
(equal (v-adder-carry-out c a b)
       (v-alu-nat-adder-carry-out c a b))

(equal (v-adder-output c a b)
       (v-alu-nat-adder-output c a b))
```

Let us focus further on the second of these equalities, as the first one is analogous (it is for the carry bit rather than for the bit-vector result). This equality supports our use of the term "interpretation lemma," as it relates a specification made at the bit level to one made using natural number operations. In particular, the function `V-ADDER-OUTPUT` returns the bit-level specification of the add-with-carry operation's bit-vector result, while the function `V-ALU-NAT-ADDER-OUTPUT` is specified using natural number operations. Let us look at these two functions in turn.

The function `V-ADDER` is a bit-level specification of the bit-vector returned by the ALU's adder, and the function `V-ADDER-OUTPUT` is the result of stripping the high-order (carry) bit from that result. These definitions, the first of which happens to be recursive, are as follows.

```
Definition.
(v-adder c a b)
 =
(if (nlistp a)
    (cons (boolfix c) nil)
  (cons (b-xor3 c (car a) (car b))
        (v-adder (b-or (b-and (car a) (car b))
                       (b-or (b-and (car a) c)
                             (b-and (car b) c)))
                 (cdr a)
                 (cdr b))))
```

```
Definition.
(v-adder-output c a b)
 =
(firstn (length a) (v-adder c a b))
```

On the other hand, the function `V-ALU-NAT-ADDER-OUTPUT` is written using natural number operations. It specifies what the ALU's add-with-carry operation does when the inputs are viewed as bit-vector representations of natural numbers. Informally: the inputs `C`, `A`, and `B` are first converted to natural numbers, then added together, and finally, the result is converted back to a bit vector of the appropriate length.

```
Definition.
(v-alu-nat-adder-output c a b)
 =
(nat-to-v (remainder (plus (b-to-nat c)
                           (v-to-nat a)
                           (v-to-nat b))
                     (exp 2 (length a)))
          (length a))
```

For the natural number case, we do not prove anything about the overflow bit. We do prove properties about overflow when considering the integer interpretation of the ALU operations. Let us conclude this discussion by noting that the integer interpretation theorem `V-ALU-CORRECT-INT` is similar to the theorem `V-ALU-CORRECT-NAT`. Bit vectors are viewed there as representing integers by the usual twos-complement scheme.

See Section 14.71 for the event file "alu-interpretation.events".

13.3 FLAG-INTERPRETATION

In this file we prove a number of lemmas about the `STORE-RESULTP` interpretations of the overflow, carry, zero, and negative bits flags, one for each particular kind of ALU operation when meaningful. In other words, assuming that the current instruction performs some operation, these lemmas tell how subsequent instructions would interpret the flag values computed in earlier instructions.

See Section 14.72 for the event file "flag-interpretation.events".

13.4 MORE-ALU-INTERPRETATION

The principal events of this file are the lemmas `BV-V-ALU-AS-NATURAL` and `BV-V-ALU-AS-INTEGER`. The lemmas in this file simply restate what was proved in the file "alu-interpretation.events" in an easily readable way.

See Section 14.73 for the event file "more-alu-interpretation.events".

Chapter 14

LIST OF FILES

In this chapter we present all of the source files used to define the FM9001 and check its correctness with Nqthm. Each file is given as a separate section starting on a separate page. The files are given in the order processed during an Nqthm run. Simply loading the second file, "sysdef.lisp", causes all the remaining files to be processed. The first file contains the public distribution license.

```
                        FM9001 PUBLIC SOFTWARE LICENSE
                          Computational Logic, Inc.
                          1717 West Sixth, Suite 290
                           Austin, Texas 78703-4776

Please  read this license carefully before using the FM9001 Software. By using
the FM9001 Software, you are agreeing  to  be  bound  by the  terms  of  this
license. If you do not agree to the terms of this license, promptly return the
FM9001 Software to the place where you obtained it.

The FM9001 Software was developed by Computational Logic, Inc.(CLI).  You  own
the  disk  or  other  medium on which the FM9001 Software is recorded, but CLI
retains title to the FM9001 Software.  The purposes of  this  license  are  to
identify  the  FM9001  Software and to make the FM9001 Software, including its
source code, freely  available.   This  license  allows  you  to  use,  copy,
distribute  and  modify  the FM9001 Software, on the condition that you comply
with all the Copying Policies set out below.

COPYING POLICIES

1. You may copy and distribute verbatim copies of the FM9001 Software  as  you
receive  it,  in  any  medium,  including  embedding it verbatim in derivative
works, provided that you a) conspicuously and appropriately  publish  on  each
copy a valid copyright notice "Copyright (C) 1990-1994 by Computational Logic,
Inc. All Rights Reserved.", b) keep intact on all files the notices that refer
to  this License Agreement and to the absence of any warranty, and c) give all
recipients of the FM9001 Software a copy of this License Agreement along  with
the program.
```

14.1 "sysdef.lisp"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    SYSDEF.LISP
;;;
;;;  Utilities for creating the FM9001 event libraries.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(in-package "USER")

;;;  This form loads all of the Common Lisp support files.  This form must be
;;;  evaluated before staring from a "NOTE-LIB".
```

```
(progn
  (backquote-setting 'nqthm)

  ;;  Akcl specific settings.  The following form may be necessary depending
  ;;  on the size of the Akcl.
  ;;  #+akcl  (setf si::*multiply-stacks* 4)
  ;;  #+akcl  (setf si::*notify-gbc* nil)
  ;;  #+(and akcl sparc) (setq compiler::*split-files* 200000)

  ;;  Lisp Works specific settings.
  ;;  #+LISPWORKS (lw::extend-current-stack 1000)

  ;;  Lucid specific settings.
  ;; #+Lucid (CHANGE-MEMORY-MANAGEMENT :GROWTH-LIMIT 2000)

  ;;  Work around a bug in Allegro.  This is a work around for the '(if* . t)
  ;;  PRINT bug in Allegro CL 4.1 suggested by duane@Franz.COM, concering
  ;;  bug report spr6115:
  ;; #+Allegro (SET-PPRINT-DISPATCH '(CONS (MEMBER IF*)) NIL)

  (setq *thm-suppress-disclaimer-flg* t)
  (setq reduce-term-clock 2000)
  ;;  (setf linel-value 69)

  ;; Following used in DO-FILES-WITH-INTERMEDIATE-LIBS below:
  (load "do-files.lisp"            :print t)

  ;; Following used in approx.events (via monotonicity-macros.lisp):
  (load "do-events-recursive.lisp" :print t)

  (load "disable.lisp"             :print t)
  (load "macros.lisp"              :print t)
  (load "expand.lisp"              :print t)
  (load "vector-macros.lisp"       :print t)
  (load "primp-database.lisp"      :print t)
  (load "primitives.lisp"          :print t)
  (load "control.lisp"             :print t)
  (load "expand-fm9001.lisp"       :print t)
  (load "monotonicity-macros.lisp" :print t)
  (load "translate.lisp"           :print t)
  (load "purify.lisp"              :print t))

;;;  Below we define a load sequence for the FM9001 specification and proof.
;;;  This sequence creates a number of intermediate libraries.  If any failures
;;;  occur along the way, a library called "failed" will be created.  If the
;;;  run was a success, one should go back and delete the intermediate
;;;  libraries since they take up a lot of space.
```

```
(defmacro do-files-with-intermediate-libs (args)
  (if args
      `(IF (DO-FILES ',(caar args))
           (PROGN
            (MAKE-LIB ,(cadar args) T)
            (DO-FILES-WITH-INTERMEDIATE-LIBS ,(cdr args)))
         (MAKE-LIB "failed"))
    nil))
```

```
(time
 (do-files-with-intermediate-libs
  ((("bags.events"
     "naturals.events"
     "integers.events"
     "math-disable.events"
     "intro.events"
     "list-rewrites.events"
     "indices.events"
     "hard-specs.events"
     "value.events"
     "memory.events"
     "dual-port-ram.events"
     "fm9001-memory.events"
     "tree-number.events"
     "f-functions.events"
     "dual-eval.events"
     "predicate-help.events"
     "predicate-simple.events"
     "predicate.events"
     "primitives.events"
     "unbound.events"
     "vector-module.events"
     "translate.events")        "dual-eval")
   (("examples.events"
     "example-v-add.events"
     "pg-theory.events"
     "tv-if.events"
     "t-or-nor.events"
     "fast-zero.events"
     "v-equal.events"
     "v-inc4.events"
     "tv-dec-pass.events"
     "reg.events"
     "alu-specs.events"
     "pre-alu.events"
     "tv-alu-help.events"
     "post-alu.events"
     "core-alu.events"
     "fm9001-spec.events"
     "asm-fm9001.events"
     "store-resultp.events"
     "control-modules.events"
     "control.events"
     "regfile.events"
     "flags.events"
     "extend-immediate.events"
     "pad-vectors.events"
     "fm9001-hardware.events"
     "chip.events")             "chip")
   (("expand-fm9001.events"
     "proofs.events"
     "approx.events"
     "final-reset.events"
```

```
     "well-formed-fm9001.events")      "proofs")
   (("math-enable.events"
     "alu-interpretation.events"
     "flag-interpretation.events"
     "more-alu-interpretation.events")  "fm9001"))))

;;;  This form creates the "clean" version of the events.  See "purify.lisp".

;  (time (library-to-events "fm9001"))

;;;  This form reruns the "clean" version of the events.

;  (time (prove-file-out "fm9001-replay"))
```

14.2 "sysload.lisp"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    SYSLOAD.LISP
;;;
;;;  Utility file to load Common Lisp macros.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(in-package "USER")

;;;  This loads all of the Common Lisp support files.  This form must be
;;;  evaluated before staring from a "NOTE-LIB".
```

```
(progn
  (backquote-setting 'nqthm)

  ;;  Akcl specific settings.  The following form may be necessary depending
  ;;  on the size of the Akcl.
  ;;  #+akcl  (setf si::*multiply-stacks* 4)
  ;;  #+akcl  (setf si::*notify-gbc* nil)
  ;;  #+(and akcl sparc) (setq compiler::*split-files* 200000)

  ;;  Lisp Works specific settings.
  ;;  #+LISPWORKS (lw::extend-current-stack 1000)

  ;;  Lucid specific settings.
  ;; #+Lucid (CHANGE-MEMORY-MANAGEMENT :GROWTH-LIMIT 2000)

  ;;  Work around a bug in Allegro.  This is a work around for the '(if* . t)
  ;;  PRINT bug in Allegro CL 4.1 suggested by duane@Franz.COM, concering
  ;;  bug report spr6115:
  ;; #+Allegro (SET-PPRINT-DISPATCH '(CONS (MEMBER IF*)) NIL)

  (setq *thm-suppress-disclaimer-flg* t)
  (setq reduce-term-clock 2000)
  ;;  (setf linel-value 69)

  ;; Following used in DO-FILES-WITH-INTERMEDIATE-LIBS below:
  (load "do-files.lisp"            :print t)

  ;; Following used in approx.events (via monotonicity-macros.lisp):
  (load "do-events-recursive.lisp" :print t)

  (load "disable.lisp"             :print t)
  (load "macros.lisp"              :print t)
  (load "expand.lisp"              :print t)
  (load "vector-macros.lisp"       :print t)
  (load "primp-database.lisp"      :print t)
  (load "primitives.lisp"          :print t)
  (load "control.lisp"             :print t)
  (load "expand-fm9001.lisp"       :print t)
  (load "monotonicity-macros.lisp" :print t)
  (load "translate.lisp"           :print t)
  (load "purify.lisp"              :print t))

;;;  Below we define a load sequence for the FM9001 specification and proof.
;;;  This sequence creates a number of intermediate libraries.  If any failures
;;;  occur along the way, a library called "failed" will be created.  If the
;;;  run was a success, one should go back and delete the intermediate
;;;  libraries since they take up a lot of space.
```

```
(defmacro do-files-with-intermediate-libs (args)
  (if args
      `(IF (DO-FILES ',(caar args))
           (PROGN
            (MAKE-LIB ,(cadar args) T)
            (DO-FILES-WITH-INTERMEDIATE-LIBS ,(cdr args)))
         (MAKE-LIB "failed"))
    nil))


(princ "
The FM9001 macros have been defined.  You should next execute:

   (in-package \"USER\")

and then you may execute any of:

   (note-lib \"dual-eval\" t)
   (note-lib \"chip\" t)
   (note-lib \"proofs\" t)
   (note-lib \"fm9001\" t)

")
```

14.3 "do-files.lisp"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.


(in-package "USER")


(defun safe-cadr (x)
  (and (consp x)
       (consp (cdr x))
       (cadr x)))


(defun library-filename (filename)
  (let ((pathname (pathname filename)))
    (namestring (make-pathname :host (pathname-host pathname)
                               :device (pathname-device pathname)
                               :directory (pathname-directory pathname)
                               :name (pathname-name pathname)))))


(defun make-lib-conditional (filename save? compile?)
; Do a MAKE-LIB and compile the resulting .lisp file depending on
; the value of the flags SAVE? and COMPILE?
  (cond (save? (make-lib filename)
               (cond (compile?
                      (proclaim-nqthm-file filename)
                      (compile-file
                       (extend-file-name filename
                                         file-extension-lisp)))))))


(defun do-files (infiles &optional (save nil) (compile nil))
  ;; This does a sequence of files, stopping the first time a failed
  ;; event is encountered. If the flag SAVE is set, a library is saved
  ;; at the end of each file. If the flag SAVE is set and the flag COMPILE
  ;; is set, then the saved lisp file is compiled.
  (if (every #'probe-file infiles)
      (iterate for infile in infiles
               do (cond ((do-file infile)
                         (make-lib-conditional (library-filename infile)
                                               save compile))
                        (t (make-lib-conditional (library-filename infile)
                                                 save compile)
                           (return nil)))
               finally (return t))
      (format t "~%*** File ~a not found. ***"
              (some (function (lambda (x) (and (not (probe-file x)) x)))
                    infiles))))
```

```
(DEFUN DO-FILE (INFILE &optional start-name)

;; patched so that one can specify the event at which one wants to begin

;   This function executes each of the event commands in the file INFILE.
;   The events are top level forms in the file.  It prints
;   each event form to PROVE-FILE and then executes it, accumulating the total
;   event times and printing the event names to the terminal if the output is
;   going elsewhere.  It aborts if some event causes an error or fails.  It
;   prints the system configuration and the accumulated times at the end of
;   PROVE-FILE.  It returns T if all events succeeded and NIL if some failed.

  (WITH-OPEN-FILE (INSTREAM (EXTEND-FILE-NAME INFILE NIL)
                            :DIRECTION :INPUT
                            :IF-DOES-NOT-EXIST :ERROR)
    (LET (ANS FORM)
      (PROG NIL

            (when start-name
                  (loop  (setq form (READ INSTREAM NIL A-VERY-RARE-CONS))
                         (COND ((EQ FORM A-VERY-RARE-CONS)
                                (RETURN T)))
                         (when (eq (safe-cadr form) start-name)
                               (go run-form))))
            LOOP
            (SETQ FORM (READ INSTREAM NIL A-VERY-RARE-CONS))
            run-form

            (COND ((EQ FORM A-VERY-RARE-CONS)
                   (RETURN T)))

;   Print out the event form to PROVE-FILE and, if PROVE-FILE is not the
;   terminal, print the name to the terminal

            (ITERPRIN 1 PROVE-FILE)
            (IPRINC EVENT-SEPARATOR-STRING PROVE-FILE)
            (ITERPRIN 2 PROVE-FILE)
            (PPRIND FORM 0 0 PROVE-FILE)
            (ITERPRI PROVE-FILE)
            (COND ((NOT (EQ PROVE-FILE NIL))
                   (IPRINC (safe-CADR FORM) NIL)))

;   Evaluate the event form

            (SETQ ANS (ERROR1-SET (EVAL FORM)))
            (COND ((NULL ANS)

;   A SOFT ERROR1 occurred during the evaluation.  Perhaps we should
;   let the user edit the form, but we have no standard editor in the
;   system.

                   (RETURN NIL)))

;   Recover the actual value from the CONS produced by the ERROR1-SET
```

```
;   protection

            (SETQ ANS (CAR ANS))

;   Print the answer to PROVE-FILE and, if PROVE-FILE is not the terminal,
;   print a comma or the failure message, as appropriate, to the terminal
;   to indicate completion of the event.

            (ITERPRI PROVE-FILE)
            (IPRINC ANS PROVE-FILE)
            (COND ((NOT (EQ PROVE-FILE NIL))
                   (COND ((EQ ANS NIL)
                          (ITERPRI NIL)
                          (IPRINC FAILURE-MSG NIL)
                          (ITERPRI NIL))
                         (T (IPRINC (QUOTE |,|) NIL)
                            (COND ((< (OUR-LINEL NIL NIL)
                                      (IPOSITION NIL NIL NIL))
                                   (ITERPRI NIL)))))))

;   Exit if the command failed.

            (COND ((EQ ANS NIL) (RETURN NIL)))

;   Otherwise, continue looping.

            (GO LOOP)))))
```

14.4 "do-events-recursive.lisp"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.


(in-package "USER")


(defun do-events-recursive (ev)
  (let (undone-events)
    (do-events ev)))
```

14.5 "disable.lisp"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    DISABLE.LISP
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(in-package "USER")


(defun disableable-citizens (event-name bad-names)
  (iterate for x in (cons event-name (get event-name 'satellites))
           when (and (not (member-eq x bad-names))
                     (not (disabledp x)))
           collect x))


(defun enabled-citizens-back-to (name)
  (or (member-eq name chronology)
      (er soft (name) (!ppr name nil) |is|
          |not| |in| |the| |chronology| excl))
  (iterate for x in chronology
           with bad-names =
           ;; from definition in nqthm of CHK-DISABLEABLE
           (append
            (iterate for x in *1*btm-objects collect
                     (pack (list prefix-for-functions x)))
            (iterate for x in shell-alist collect
                     (pack (list prefix-for-functions (car x)))))
           until (eq x name)
           when (not (member-eq (car (get x 'event)) '(toggle deftheory)))
           nconc (disableable-citizens x bad-names)))
```

```
(defun enabled-citizens-between (start-name end-name &aux temp)
  (or (member-eq start-name chronology)
      (er soft (start-name) (!ppr start-name nil) |is|
          |not| |in| |the| |chronology| excl))
  (or (setq temp (member-eq end-name chronology))
      (er soft (end-name) (!ppr end-name nil) |is|
          |not| |in| |the| |chronology| excl))
  (or (member-eq start-name temp)
      (er soft (start-name end-name) (!ppr start-name nil) |is|
          |in| |the| |chronology| |,| |but| |occurs| |later|
          |than| (!ppr end-name nil) excl))
  (iterate for x in (member-eq end-name chronology)
           with last-name
           and bad-names =
           ;; from definition in nqthm of CHK-DISABLEABLE
           (append
            (iterate for x in *1*btm-objects collect
                     (pack (list prefix-for-functions x)))
            (iterate for x in shell-alist collect
                     (pack (list prefix-for-functions (car x)))))
           until (prog1 (eq (print last-name) (print start-name))
                   (setq last-name x))
           when (not (member-eq (car (get x 'event)) '(toggle deftheory)))
           nconc (disableable-citizens x bad-names)))


(defun disable-all-fn (lst)
  (cons 'progn
        (iterate for x in lst
                 collect (list 'disable x))))


(defmacro disable-all (lst &optional (name 'temp))
  (disable-all-fn lst))


(defun enable-all-fn (lst)
  (cons 'progn
        (iterate for x in lst
                 collect (list 'enable x))))


(defmacro enable-all (&rest lst)
  (enable-all-fn lst))
```

```
(defmacro disable-back-to (name &optional theory-name print-flg)
  (let* ((enabled-citizens (enabled-citizens-back-to name))
         (disabling-events
          (iterate for x in enabled-citizens
                   collect (list 'disable x)))
         (form
          (cons 'progn
                (if theory-name
                    (cons `(deftheory ,theory-name ,enabled-citizens)
                          disabling-events)
                  disabling-events))))
    (if print-flg
        (print form)
      form)))

;; To reproduce the existing enable-disable state, we disable all
;; non-disabled new disableable events, and we also restore all
;; existing events that had been touched.


(defun disabledp-as-of (name alternate-disabled-lemmas)
  (let ((pair (assoc-eq name alternate-disabled-lemmas)))
    (if pair
        (cddr pair)
      nil)))


(defun untampering-events (name)
  (let ((alternate-disabled-lemmas disabled-lemmas)
        possibly-tampered-names)
    ;; take new events off alternate-disabled-lemmas and
    ;; set possibly-tampered-names
    (iterate for x in chronology
             until (or (eq x name)
                       (null alternate-disabled-lemmas))
             when (eq (cadr (car alternate-disabled-lemmas)) x)
             do
             (setq possibly-tampered-names
                   (add-to-set (car (car alternate-disabled-lemmas))
                               possibly-tampered-names))
             (setq alternate-disabled-lemmas
                   (cdr alternate-disabled-lemmas)))
    (and (not (eq alternate-disabled-lemmas disabled-lemmas))
         (iterate for x in possibly-tampered-names
                  for old-status = (disabledp-as-of
                                    x alternate-disabled-lemmas)
                  when (not (eq old-status (disabledp x)))
                  collect (if old-status
                              `(disable ,x)
                            `(enable ,x))))))
```

```
(defmacro better-disable-back-to (name &optional print-flg)
  (let* ((enabled-citizens (enabled-citizens-back-to name))
         (disabling-events
          (iterate for x in enabled-citizens
                   collect (list 'disable x)))
         (form
          (cons 'progn
                (append disabling-events (untampering-events name)))))
    (if print-flg
        (print form)
      form)))


(defmacro disable-closed-interval (start-name end-name
                                              &optional theory-name print-flg)
  (let* ((enabled-citizens (enabled-citizens-between start-name end-name))
         (disabling-events
          (iterate for x in enabled-citizens
                   collect (list 'disable x)))
         (form
          (cons 'progn
                (if theory-name
                    (cons `(deftheory ,theory-name ,enabled-citizens)
                          disabling-events)
                  disabling-events))))
    (if print-flg
        (print form)
      form)))


(defmacro enable-or-disable-theory (name enable-flg print-flg)
  (let (citizen-names)
    (or (match (get name 'event)
               (deftheory & citizen-names))
        (er soft (name) |There| |is| |no| DEFTHEORY |event|
            |called| (!ppr name (quote |.|))))
    (let ((form (cons 'progn
                      (iterate for x in citizen-names
                               with ev-type = (if enable-flg 'enable 'disable)
                               when (eq (disabledp x) enable-flg)
                               collect (list ev-type x)))))
      (if print-flg
          (print form)
        form))))


(defmacro disable-theory (name &optional print-flg)
  `(enable-or-disable-theory ,name nil ,print-flg))


(defmacro enable-theory (name &optional print-flg)
  `(enable-or-disable-theory ,name t ,print-flg))
```

14.6 "macros.lisp"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    MACROS.LISP
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(in-package "USER")

;;  We use this function to eliminate unused variable messages from
;;  within structured "LET's" inside backquoted expresions.


(defun ignore-variable (x) x)

;;;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    UNBREAK
;;;
;;;    Turn off rewrite tracing.
;;;
;;;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defun unbreak ()
  (unbreak-lemma)
  (maintain-rewrite-path nil)
  (format t "~%Ignore the above message about the rewrite path being ~
              maintained."))

;;;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    NQFIX
;;;    NQNULL
;;;    NQCAR
;;;    NQCDR
;;;    NQNTH
;;;
;;;    These are for destructuring BM terms as they are stored on the property
;;;    lists in the NQTHM database.
;;;
;;;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defun nqfix (form)
  (list 'QUOTE form))
```

```
(defun nqnull (form)
  (and (equal (car form) 'QUOTE)
       (null (cadr form))))


(defun nqcar (form)
  (if (equal (car form) 'QUOTE)
      (nqfix (caadr form))
    (if (equal (car form) 'CONS)
        (cadr form)
      (if (equal (car form) 'APPEND)
          (if (nqnull (cadr form))
              (nqcar (caddr form))
            (nqcar (cadr form)))
        (error "~s" form)))))


(defun nqcdr (form)
  (if (equal (car form) 'QUOTE)
      (nqfix (cdadr form))
    (if (equal (car form) 'CONS)
        (caddr form)
      (if (equal (car form) 'APPEND)
          (if (nqnull (cadr form))
              (nqcdr (caddr form))
            `(APPEND ,(nqcdr (cadr form)) ,(caddr form)))
        (error "~s" form)))))


(defun nqnth (n form)
  (if (zerop n)
      (nqcar form)
    (nqnth (1- n) (nqcdr form))))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    UNSTRING &rest args
;;;
;;;  Concatenates the STRING of each of its args (converting numbers to decimal
;;;  form), and INTERNs the result, returning the symbol.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defun unstring (&rest args)
  (intern (apply #'concatenate 'string
                 (mapcar #'(lambda (x)
                             (if (numberp x)
                                 (format nil "~d" x)
                               (string x)))
                         args))))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    SUBMODULES generator &key body
;;;    SUBMODULE-VALUE-LEMMAS generator &key body
;;;
;;;    Looks inside a generator and creates a list of the referenced
;;;    submodules.  The :BODY keyword gives a "side-door" entrance for cases
;;;    where the GENERATOR has not actually been stored, but we know what the
;;;    body is (see DEFN-TO-MODULE).
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defun submodules (generator &key body)
  (let* ((event (get generator 'event))
         (form (nth 3 event))
         (body (if body body (nqnth 3 form))))
    (labels
      ((collect
        (l)
        (cond
         ((nqnull l) nil)
         (t (cons (nqnth 2 (nqcar l)) (collect (nqcdr l)))))))
      (remove-duplicates (mapcar #'cadr (collect body))))))


(defun submodule-value-lemmas (generator &key body)
  (mapcar #'(lambda (x) (unstring x "$VALUE"))
          (submodules generator :body body)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    MODULE-PREDICATE generator
;;;
;;;  MODULE-PREDICATE writes the netlist predicate for simple modules.  For
;;;  example, if the module is defined by (M*), then (MODULE-PREDICATE M*)
;;;  creates (M& NETLIST).
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defmacro module-predicate (generator)
  (let* ((event (get generator 'event))
         (form (nth 3 event))
         (name (cadr (nqcar form)))
         (fn& (unstring name "&"))
         (submodules (submodules generator))
         (sub& (mapcar #'(lambda (x)
                           `(,(unstring x "&")
                             (DELETE-MODULE ',name NETLIST)))
                       submodules)))

    `(PROGN

      (DEFN ,fn& (NETLIST)
        (AND (EQUAL (LOOKUP-MODULE ',name NETLIST) (,generator))
             ,@sub&))

      (DISABLE ,fn&))))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    MODULE-NETLIST generator
;;;
;;;  MODULE-NETLIST assembles the netlist for simple modules.  For
;;;  example, if the module is defined by (M*), then (MODULE-NETLIST M*)
;;;  creates (M$NETLIST NETLIST).
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defmacro module-netlist (generator)
  (let* ((event (get generator 'event))
         (form (nth 3 event))
         (name (cadr (nqcar form)))
         (netfn (unstring name "$NETLIST"))
         (submodules (submodules generator))
         (subnetlists (mapcar #'(lambda (x) `(,(unstring x "$NETLIST")))
                              submodules))
         (body (reduce #'(lambda (x y) `(UNION ,x ,y)) subnetlists)))

    `(DEFN ,netfn () (CONS (,generator) ,body))))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    DEFN-TO-MODULE
;;;
;;;    DEFN-TO-MODULE converts simple BM specifications to DUAL-EVAL constants.
;;;    The BM specifications may be multiple output, but every referenced
;;;    primitive must be single output.  No vectors on inputs, outputs, or
;;;    internal modules are allowed.  Examples abound in the events.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defvar signal-counter 0)

(defvar gate-counter 0)


(defun gensignal ()
  (prog1 (unstring "W-" signal-counter) (incf signal-counter)))

(defun gengate ()
  (prog1 (unstring "G-" gate-counter) (incf gate-counter)))


(defun expand-body (body &optional top)
  (cond
   ((symbolp body) (if top
                       (let ((name (list (gensignal))))
                         (values name
                                 (list (list (gengate) name 'ID (list body)))))
                     (values (list body) nil)))
   ((and (equal (car body) 'quote) (null (cadr body))) (values nil nil))
   ((equal (car body) 'cons)
    (multiple-value-bind (car-outputs car-body) (expand-body (cadr body) t)
      (multiple-value-bind (cdr-outputs cdr-body) (expand-body (caddr body))
        (values (append car-outputs cdr-outputs)
                (append car-body cdr-body)))))
   (t (let* ((fn (car body))
             (x-args (mapcar #'(lambda (x)
                                 (multiple-value-list (expand-body x)))
                             (cdr body)))
             (wires (mapcar #'caar x-args))
             (subbody (mapcan #'cadr x-args))
             (name (list (gensignal))))

        (values name (cons (list (gengate) name fn wires) subbody))))))
```

```
(defun compress-body (outputs body r-body)
  (cond
   ((null body) (values outputs (reverse r-body)))
   (t (let* ((occ (car body))
             (lhs (cadr occ))
             (fn  (caddr occ))
             (rhs (cadddr occ))
             (tail (member 0 (cdr body)
                          :test #'(lambda (x occ) (declare (ignore x))
                                    (and (equal (caddr occ) fn)
                                         (equal (cadddr occ) rhs))))))
        (cond
         ((not tail) (compress-body outputs (cdr body) (cons occ r-body)))
         (t (let* ((new-occ (car tail))
                   (new-lhs (cadr new-occ))
                   (alist (mapcar #'cons new-lhs lhs)))
              (compress-body
               (sublis alist outputs)
               (cons occ (sublis alist
                                 (delete new-occ (cdr body) :test #'equal)))
               r-body))))))))


(defun fixup-duplicate-outputs (outputs body)
  (let ((outputs
         (iterate for (output . rest) on outputs
           collecting
           (if (member output rest)
               (let* ((new-output (gensignal))
                      (new-form `(,(gengate) (,new-output) ID (,output))))
                 (setf body (append body (list new-form)))
                 new-output)
             output))))
    (values outputs body)))


(defun logic-to-hdl (body)
  (setf signal-counter 0)
  (setf gate-counter 0)
  (multiple-value-bind (outputs body)
    (expand-body body t)
    (multiple-value-bind (outputs body)
      (compress-body outputs (reverse body) nil)
      (multiple-value-bind (outputs body)
        (fixup-duplicate-outputs outputs body)
        (values outputs body)))))
```

```
(defun b-to-f (name)
  (if (member name '(ID VSS VDD))
      name
    (let ((string (string name)))
      (if (equal (subseq string 0 2) "B-")
          (unstring "F-" (subseq string 2))
        (unstring "F$" string)))))


(defun f-body (body)
  (cond
   ((symbolp body) body)
   ((and (equal (car body) 'quote) (null (cadr body))) NIL)
   ((equal (car body) 'cons)
    `(CONS ,(f-body (cadr body)) ,(f-body (caddr body))))
   (t (cons (b-to-f (car body))
            (mapcar #'(lambda (x) (f-body x)) (cdr body))))))
```

```
(defmacro defn-to-module (spec &key value-lemma boolp-value-lemma)
  (let* ((defn (get spec 'event))
         (name (second defn))
         (inputs (third defn))
         (body (fourth defn)))
    (multiple-value-bind (hdl-outputs hdl-body)
      (logic-to-hdl body)
      (let* ((f-body (f-body body))
             (f$name (unstring "F$" name))
             (fn* (unstring name "*"))
             (fn& (unstring name "&"))
             (value-lemma (if value-lemma
                              value-lemma
                            (unstring name "$VALUE")))
             (boolp-value-lemma
              (if boolp-value-lemma
                  boolp-value-lemma
                (unstring f$name "=" name)))
             (boolp-inputs (mapcar #'(lambda (x)
                                       `(BOOLP ,x))
                                   inputs)))

        `(PROGN

          (DEFN ,f$name ,inputs ,f-body)

          (DISABLE ,f$name)

          (DEFN ,fn* () '(,name ,inputs ,hdl-outputs ,hdl-body nil))

          (MODULE-PREDICATE ,fn*)

          (MODULE-NETLIST ,fn*)

          (PROVE-LEMMA ,value-lemma (REWRITE)
            (IMPLIES
             (,fn& NETLIST)
             (EQUAL (DUAL-EVAL 0 ',name (LIST ,@inputs) STATE NETLIST)
                    ,(if (null (cdr hdl-outputs))
                         `(LIST (,f$name ,@inputs))
                       `(,f$name ,@inputs))))
            ;;Hint
            ((ENABLE ,fn& ,f$name
                     ,@(submodule-value-lemmas fn* :body (nqfix hdl-body)))
             (DISABLE-THEORY F-GATES)))

          (DISABLE ,value-lemma)

          (PROVE-LEMMA ,boolp-value-lemma (REWRITE)
            (IMPLIES
             ,(if (null (cdr boolp-inputs))
                  (car boolp-inputs)
                `(AND ,@boolp-inputs))
             (EQUAL (,f$name ,@inputs)
```

```
                   (,name ,@inputs)))
            ;;Hint
            ((ENABLE BOOLP-B-GATES ,f$name ,spec)
             (DISABLE-THEORY F-GATES B-GATES)))

          )))))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    DESTRUCTURING-LEMMA generator
;;;
;;;  Because of quirks in equality reasoning, it "doesn't work" to simply let
;;;  module definitions open up.  Instead, we use a lemma that explicitly
;;;  states how to destructure a module definition.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defmacro destructuring-lemma (generator)
  (let* ((event (get generator 'event))
         (args (third event))
         (defn (fourth event))
         (name (nqnth 0 defn))
         (inputs (nqnth 1 defn))
         (outputs (nqnth 2 defn))
         (body (nqnth 3 defn))
         (states (nqnth 4 defn))
         (form `(,generator ,@args))
         (destructuring-lemma (unstring generator "$DESTRUCTURE")))
    `(PROGN

      (PROVE-LEMMA ,destructuring-lemma (REWRITE)
        (AND
         (EQUAL (CAR ,form) ,name)
         (EQUAL (CADR ,form) ,inputs)
         (EQUAL (CADDR ,form) ,outputs)
         (EQUAL (CADDDR ,form) ,body)
         (EQUAL (CADDDDR ,form) ,states)))

      (DISABLE ,generator)

      (DISABLE ,destructuring-lemma))))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    MODULE-GENERATOR generator args name inputs outputs body state
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defmacro module-generator ((generator . args) name inputs outputs body state)
  (let ((destructuring-lemma (unstring generator "$DESTRUCTURE"))
        (form `(,generator ,@args)))

    `(PROGN

      (DEFN ,generator ,args
        (LIST ,name ,inputs ,outputs ,body ,state))

      (PROVE-LEMMA ,destructuring-lemma (REWRITE)
        (AND
         (EQUAL (CAR ,form) ,name)
         (EQUAL (CADR ,form) ,inputs)
         (EQUAL (CADDR ,form) ,outputs)
         (EQUAL (CADDDR ,form) ,body)
         (EQUAL (CADDDDR ,form) ,state)))

      (DISABLE ,generator)

      (DISABLE ,destructuring-lemma))))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    GENERATE-BODY-INDUCTION-SCHEME generator
;;;
;;;   Creates a DUAL-EVAL 1 induction scheme for body generators of the form
;;;
;;;   (IF test something (CONS occurrence recursive-call))
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defmacro generate-body-induction-scheme (generator)
  (let* ((event (get generator 'event))
         (name (unstring generator "$INDUCTION"))
         (call `(,generator ,@(nth 2 event)))
         (args (append (nth 2 event) '(BINDINGS STATE-BINDINGS NETLIST)))
         (body (nth 3 event))
         (test (nth 1 body))
         (recursion (cdr (nqcdr (nth 3 body)))))

    `(DEFN ,name ,args
       (IF ,test
           t
         (,name ,@recursion
                (DUAL-EVAL-BODY-BINDINGS 1 ,call
                                         BINDINGS STATE-BINDINGS NETLIST)
                STATE-BINDINGS
                NETLIST)))))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    #V reader macro.
;;;
;;;  A reader macro for bit-vectors. For example, #v001 = (LIST T F F).
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defun bit-vector-reader (stream subchar arg)
  ;;  We "unread" the vector character, and reread to get a symbol.  Otherwise
  ;;  the number following the vector character might be read as a leading zero
  ;;  of an integer.
  (declare (ignore subchar arg))
  (unread-char #\v stream)
  (let ((symbol (read stream t nil t)))
    ;;  Get rid of the vector character, reverse, and list for NQTHM.
    `(LIST ,@(map 'list #'(lambda (x)
                            (if (equal x #\1)
                                't
                              (if (equal x #\0)
                                  'f
                                (error "Non-binary digits in --> ~s."
                                       symbol))))
                  (reverse (subseq (symbol-name symbol) 1))))))
```

```
(eval-when (load eval)
  (set-dispatch-macro-character #\# #\v #'bit-vector-reader))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    #I reader macro.
;;;
;;;  A reader macro for indices.
;;;  For example, #i(a 10) = (INDEX 'A 10), and #i(a 0 n) = (INDICES 'A 0 10).
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defun index-reader (stream subchar arg)
  (declare (ignore subchar arg))
  (let* ((args (read stream t nil t))
         (name (car args))
         (num  (cadr args))
         (n    (caddr args)))
    (if n
        `(INDICES ',name ,num ,n)
      `(INDEX ',name ,num))))


(eval-when (load eval)
  (set-dispatch-macro-character #\# #\i #'index-reader))
```

14.7 "expand.lisp"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    EXPAND.LISP
;;;
;;;    EXPAND is a way to use the NQTHM rewriter to rewrite terms, for example
;;;    to provide the right hand side of complicated lemmas.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(in-package "USER")
```

```
(defun expand (form &optional (hyps '(true)) hints)
  (chk-init)
  (let ((dummy-var (gentemp))
        term clauses answer last-literal lhs rhs expansion)
    ;;  Check the term and hints.
    (setf term (translate-and-chk `(IMPLIES ,hyps (EQUAL ,form ,dummy-var))))
    (match! (chk-acceptable-hints hints)
            (list hints))
    (unless (iterate for hint in hints
              always (member (car hint)
                             '(enable disable enable-theory disable-theory
                                      no-built-in-arith expand hands-off)))
      (error "The only allowable hints are ENABLE, DISABLE, ENABLE-THEORY, ~
              DISABLE-THEORY, NO-BUILT-IN-ARITH and HANDS-OFF."))
    ;;  We now simplify.
    (unwind-protect
      (progn
        (setf term (apply-hints hints term))
        (setf clauses (preprocess term))
        (unless (equal (length clauses) 1)
          (error "Clausification of the original input ~
                  returned multiple clauses. ~%~s" clauses))
        (setup form clauses abbreviations-used)
        (simplify-clause-maximally (car clauses))
        (when (and (not (equal (length process-clauses) 1))
                   (not (iterate for (clause . rest) on process-clauses
                          when (consp rest)
                          always (equal (car (last clause))
                                        (car (last (car rest)))))))
          (format t
                  "Simplification returned multiple (~d) ~
                   unequivalent clauses.~%" (length process-clauses))
          (iterate for clause in process-clauses
            do (progn (pprint (prettyify-clause clause)) (terpri)))
          (error "Too many clauses."))
        (setf answer (car process-clauses))
        (setf last-literal (car (last answer)))
        (unless (and (match last-literal (equal lhs rhs))
                     (or (progn (setf expansion lhs)
                                (eq rhs dummy-var))
                         (progn (setf expansion rhs)
                                (eq lhs dummy-var))))
          (error "The resulting clause does not have the expected form :~
                  ~%~s" (prettyify-clause answer)))
        (untranslate expansion))
      (iterate for x in hint-variable-alist
        do (set (cadr x) (cadddr x))))))
```

```
(defmacro expand-lemma (name type hyps term &optional hints)
  (let ((expansion (expand term hyps hints)))
    (print `(PROVE-LEMMA ,name ,type
              ,(if (null hyps)
                   `(EQUAL ,term ,expansion)
                 `(IMPLIES
                   ,hyps
                   (EQUAL ,term ,expansion)))
              ,hints))))
```

14.8 "vector-macros.lisp"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    VECTOR-MACROS.LISP
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(in-package "USER")

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    VECTOR-MODULE name (occ-name outputs type inputs) specs &key enable
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    VECTOR-MODULE creates simple, linear, n-bit module generators.
;;;
;;;   Arguments:
;;;
;;;     name -- The generator will be (<name>* n)
;;;
;;;     (occ-name outputs type inputs) --  A schematic representation of the
;;;     occurrences.  The body of the generator will contain occurrences of the
;;;     form:
;;;           (<occ-name>_n
;;;            (<output_0>_n ... <output_k>_n)
;;;            type
;;;            (<input_0>_n ... <input_k>_n))
;;;
;;;     specs -- A list of specifications for the output vectors, written in
;;;     terms of the inputs.
;;;
;;;     enable -- A list of events to be enabled.
;;;
;;;   Example: (vector-module v-buf (g (y) b-buf (a)) ((v-threefix a))
;;;                           :enable (f-buf))
;;;
;;;   More examples in "vector-module.events".
```

```
(defmacro vector-module (name (occ-name outputs type inputs) specs
                              &key enable)
  (let* ((body-defn (unstring name "$BODY"))
         (generator (unstring name "*"))
         (destructor (unstring generator "$DESTRUCTURE"))
        (module-name `(INDEX ',name N))
        (predicate (unstring name "&"))
        (type-predicate (unstring type "&"))
        (unbound-in-body-lemma (unstring name "$UNBOUND-IN-BODY"))
        (body-value-lemma (unstring name "$BODY-VALUE"))
        (type-value-lemma (unstring type "$VALUE"))
        (value-lemma (unstring name "$VALUE"))
        (netlist (unstring name "$NETLIST"))
        (type-netlist (unstring type "$NETLIST")))


    (labels ((mapAPPEND
              (x)
              (if (consp x)
                  (if (consp (cdr x))
                      `(APPEND ,(car x) ,(mapAPPEND (cdr x)))
                    (car x))
                nil))
             (mapAND
              (x)
              (if (consp x)
                  (if (consp (cdr x))
                      `(AND ,(car x) ,(mapAND (cdr x)))
                    (car x))
                nil)))

      `(PROGN

        (DEFN ,body-defn (M N)
          (IF (ZEROP N)
              NIL
            (CONS
             (LIST (INDEX ',occ-name M)
                   (LIST ,@(mapcar #'(lambda (x) `(INDEX ',x M)) outputs))
                   ',type
                   (LIST ,@(mapcar #'(lambda (x) `(INDEX ',x M)) inputs)))
             (,body-defn (ADD1 M) (SUB1 N)))))

        (DISABLE ,body-defn)

        (MODULE-GENERATOR
         (,generator N)
         ,module-name
         ,(mapAPPEND
           (mapcar #'(lambda (x) `(INDICES ',x 0 N)) inputs))
         ,(mapAPPEND
           (mapcar #'(lambda (x) `(INDICES ',x 0 N)) outputs))
         (,body-defn 0 N)
         NIL)
```

```
(DEFN ,predicate (NETLIST N)
  (AND (EQUAL (LOOKUP-MODULE ,module-name NETLIST)
              (,generator N))
       (,type-predicate (DELETE-MODULE ,module-name NETLIST))))

(DISABLE ,predicate)

(DEFN ,netlist (N)
  (CONS (,generator N) (,type-netlist)))

(PROVE-LEMMA ,unbound-in-body-lemma (REWRITE)
  (IMPLIES
   (LESSP L M)
   ,(mapAND (mapcar #'(lambda (x)
                        `(UNBOUND-IN-BODY (INDEX ',x L)
                                          (,body-defn M N)))
                    outputs)))
  ;;Hint
  ((enable ,body-defn UNBOUND-IN-BODY)))

(DISABLE ,unbound-in-body-lemma)

(PROVE-LEMMA ,body-value-lemma (REWRITE)
  (IMPLIES
   (AND (,type-predicate NETLIST)
        (EQUAL BODY (,body-defn M N)))
   ,(mapAND
     (mapcar
      #'(lambda (x y)
          `(EQUAL
            (COLLECT-VALUE
             (INDICES ',x M N)
             (DUAL-EVAL 1 BODY
                        BINDINGS STATE-BINDINGS NETLIST))
            ,(let* ((spec y)
                    (fn (car spec))
                    (args (cdr spec)))
               `(,fn ,@(mapcar #'(lambda (x)
                                   `(COLLECT-VALUE
                                     (INDICES ',x M N)
                                     BINDINGS))
                               args)))))
      outputs specs)))
  ;;Hint
  ((INDUCT
    (VECTOR-MODULE-INDUCTION BODY M N BINDINGS STATE-BINDINGS NETLIST))
   (DISABLE-THEORY F-GATES)
   (ENABLE ,body-defn ,type-value-lemma ,unbound-in-body-lemma
           ,@(mapcar #'car specs)
           ,@enable)))

(DISABLE ,body-value-lemma)

(PROVE-LEMMA ,value-lemma (REWRITE)
```

```
  (IMPLIES
   (AND (,predicate NETLIST N)
        ,(mapAND
          (mapcar
           #'(lambda (x)
               `(AND (PROPERP ,x)
                     (EQUAL (LENGTH ,x) N)))
           inputs)))
   (EQUAL
    (DUAL-EVAL 0 ,module-name ,(mapAPPEND inputs)
               STATE NETLIST)
    ,(mapAPPEND
      (mapcar
       #'(lambda (spec)
           (let* ((fn (car spec))
                  (args (cdr spec)))
             `(,fn ,@args)))
       specs))))
  ;;Hint
  ((ENABLE ,predicate ,body-value-lemma ,destructor)))

(DISABLE ,value-lemma)

))))
```

14.9 "primp-database.lisp"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    PRIMP-DATABASE.LISP
;;;
;;;  We define the primitive database as a Common Lisp object so that it is
;;;  available to our netlist-to-NDL translator (see "translate.lisp"), and the
;;;  automatic primitive lemma generators (see "primitives.lisp").
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(in-package "USER")

;;;  Abbreviations


(defconstant dp-ram-16x32-inputs
  '(read-a0 read-a1 read-a2 read-a3
            write-b0 write-b1 write-b2 write-b3
            wen
            d0  d1  d2  d3  d4  d5  d6  d7
            d8  d9  d10 d11 d12 d13 d14 d15
            d16 d17 d18 d19 d20 d21 d22 d23
            d24 d25 d26 d27 d28 d29 d30 d31))


(defconstant mem-32x32-inputs
  '(rw- strobe-

        a0  a1  a2  a3  a4  a5  a6  a7
        a8  a9  a10 a11 a12 a13 a14 a15
        a16 a17 a18 a19 a20 a21 a22 a23
        a24 a25 a26 a27 a28 a29 a30 a31

        d0  d1  d2  d3  d4  d5  d6  d7
        d8  d9  d10 d11 d12 d13 d14 d15
        d16 d17 d18 d19 d20 d21 d22 d23
        d24 d25 d26 d27 d28 d29 d30 d31
        ))

;;;  CONS-UP builds a quoted expression suitable for EVAL$.
```

```
(defun cons-up (list)
  (cond
   ((null list) ''NIL)
   (t `(CONS ,(car list) ,(cons-up (cdr list))))))

;;;  The database.
```

```
(defconstant
  common-lisp-primp-database
  `((ao2
     (delays        ((263 737) (83 392)))
     (drives        10)
     (input-types   boolp boolp boolp boolp)
     (inputs        a b c d)
     (loadings      1 1 1 1)
     (lsi-name      . ao2)
     (out-depends   (a b c d))
     (output-types  boolp)
     (outputs       z)
     (results       . ,(cons-up '((f-nor (f-and a b) (f-and c d)))))

     (gates         . 2)
     (primitives    . 1)
     (transistors   . 8))



    (ao4
     (delays        ((260 895) (82 311)))
     (drives        10)
     (input-types   boolp boolp boolp boolp)
     (inputs        a b c d)
     (loadings      1 1 1 1)
     (lsi-name      . ao4)
     (out-depends   (a b c d))
     (output-types  boolp)
     (outputs       z)
     (results       . ,(cons-up '((f-nand (f-or a b) (f-or c d)))))

     (gates         . 2)
     (primitives    . 1)
     (transistors   . 8))


    (ao6
     (delays        ((260 745) (82 202)))
     (drives        10)
     (input-types   boolp boolp boolp)
     (inputs        a b c)
     (loadings      1 1 1)
     (lsi-name      . ao6)
     (out-depends   (a b c))
     (output-types  boolp)
     (outputs       z)
     (results       . ,(cons-up '((f-nor (f-and a b) c))))

     (gates         . 2)
     (primitives    . 1)
     (transistors   . 6))
```

```
(ao7
 (delays        ((261 486) (83 293)))
 (drives        10)
 (input-types   boolp boolp boolp)
 (inputs        a b c)
 (loadings      1 1 1)
 (lsi-name      . ao7)
 (out-depends   (a b c))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-nand (f-or a b) c))))

 (gates         . 2)
 (primitives    . 1)
 (transistors   . 6))


(b-and
 (delays        ((144 422) (54 707)))
 (drives        10)
 (input-types   boolp boolp)
 (inputs        a b)
 (loadings      1 1)
 (lsi-name      . an2)
 (out-depends   (a b))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-and a b))))

 (gates         . 2)
 (primitives    . 1)
 (transistors   . 6))


(b-and3
 (delays        ((146 663) (58 809)))
 (drives        10)
 (input-types   boolp boolp boolp)
 (inputs        a b c)
 (loadings      1 1 1)
 (lsi-name      . an3)
 (out-depends   (a b c))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-and3 a b c))))

 (gates         . 2)
 (primitives    . 1)
 (transistors   . 8))


(b-and4
 (delays        ((149 934) (60 870)))
 (drives        10)
```

```
 (input-types   boolp boolp boolp boolp)
 (inputs        a b c d)
 (loadings      1 1 1 1)
 (lsi-name      . an4)
 (out-depends   (a b c d))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-and4 a b c d))))

 (gates         . 3)
 (primitives    . 1)
 (transistors   . 10))


(b-equv
 (delays        ((145 742) (67 973)))
 (drives        10)
 (input-types   boolp boolp)
 (inputs        a b)
 (loadings      1 2)
 (lsi-name      . en)
 (out-depends   (a b))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-equv a b))))

 (gates         . 3)
 (primitives    . 1)
 (transistors   . 12))


(b-equv3
 (delays        ((151 1806) (79 1580)))
 (drives        10)
 (input-types   boolp boolp boolp)
 (inputs        a b c)
 (loadings      1 3 2)
 (lsi-name      . en3)
 (out-depends   (a b c))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-equv3 a b c))))

 (gates         . 7)
 (primitives    . 1)
 (transistors   . 22))



(b-if
 (delays        ((70 775) (60 1040)))
 (drives        10)
 (input-types   boolp boolp boolp)
 (inputs        s a b)
 (loadings      2 1 1)
```

```
 (lsi-name      . (mux21h b a s))   ; Note input reordering
 (out-depends   (a b s))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-if s a b))))

 (gates         . 4)
 (primitives    . 1)
 (transistors   . 12))


(b-nand
 (delays        ((141 420) (82 161)))
 (drives        10)
 (input-types   boolp boolp)
 (inputs        a b)
 (loadings      1 1)
 (lsi-name      . nd2)
 (out-depends   (a b))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-nand a b))))

 (gates         . 1)
 (primitives    . 1)
 (transistors   . 4))


(b-nand3
 (delays        ((142 537) (109 322)))
 (drives        10)
 (input-types   boolp boolp boolp)
 (inputs        a b c)
 (loadings      1 1 1)
 (lsi-name      . nd3)
 (out-depends   (a b c))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-nand3 a b c))))

 (gates         . 2)
 (primitives    . 1)
 (transistors   . 6))



(b-nand4
 (delays        ((144 588) (144 418)))
 (drives        10)
 (input-types   boolp boolp boolp boolp)
 (inputs        a b c d)
 (loadings      1 1 1 1)
 (lsi-name      . nd4)
 (out-depends   (a b c d))
 (output-types  boolp)
```

```
 (outputs       z)
 (results       . ,(cons-up '((f-nand4 a b c d))))

 (gates         . 2)
 (primitives    . 1)
 (transistors   . 8))


(b-nand5
 (delays        ((144 1002) (59 1120)))
 (drives        10)
 (input-types   boolp boolp boolp boolp boolp)
 (inputs        a b c d e)
 (loadings      1 1 1 1 1)
 (lsi-name      . nd5)
 (out-depends   (a b c d e))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-nand5 a b c d e))))

 (gates         . 4)
 (primitives    . 1)
 (transistors   . 16))


(b-nand6
 (delays        ((144 982) (59 1090)))
 (drives        10)
 (input-types   boolp boolp boolp boolp boolp boolp)
 (inputs        a b c d e g)
 (loadings      1 1 1 1 1 1)
 (lsi-name      . nd6)
 (out-depends   (a b c d e g))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-nand6 a b c d e g))))

 (gates         . 5)
 (primitives    . 1)
 (transistors   . 18))



(b-nand8
 (delays        ((144 1042) (60 1360)))
 (drives        10)
 (input-types   boolp boolp boolp boolp boolp boolp boolp boolp)
 (inputs        a b c d e g h i)
 (loadings      1 1 1 1 1 1 1 1)
 (lsi-name      . nd8)
 (out-depends   (a b c d e g h i))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-nand8 a b c d e g h i))))
```

```
 (gates         . 6)
 (primitives    . 1)
 (transistors   . 22))


(b-nbuf
 (delays        ((142 447) (57 213))
                ((143 302) (52 366)))
 (drives        9 10)
 (input-types   boolp)
 (inputs        a)
 (loadings      1)
 (lsi-name      . ivda)
 (out-depends   (a) (a))
 (output-types  boolp boolp)
 (outputs       y z)
 (results       . ,(cons-up '((f-not a) (f-buf a))))

 (gates         . 1)
 (primitives    . 1)
 (transistors   . 4))


(b-nor
 (delays        ((260 460) (59 170)))
 (drives        10)
 (input-types   boolp boolp)
 (inputs        a b)
 (loadings      1 1)
 (lsi-name      . nr2)
 (out-depends   (a b))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-nor a b))))

 (gates         . 1)
 (primitives    . 1)
 (transistors   . 4))



(b-nor3
 (delays        ((384 798) (59 224)))
 (drives        10)
 (input-types   boolp boolp boolp)
 (inputs        a b c)
 (loadings      1 1 1)
 (lsi-name      . nr3)
 (out-depends   (a b c))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-nor3 a b c))))

 (gates         . 2)
 (primitives    . 1)
```

```
 (transistors   . 6))


(b-nor4
 (delays        ((510 1072) (59 225)))
 (drives        10)
 (input-types   boolp boolp boolp boolp)
 (inputs        a b c d)
 (loadings      1 1 1 1)
 (lsi-name      . nr4)
 (out-depends   (a b c d))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-nor4 a b c d))))

 (gates         . 2)
 (primitives    . 1)
 (transistors   . 8))


(b-nor5
 (delays        ((145 1493) (55 767)))
 (drives        10)
 (input-types   boolp boolp boolp boolp boolp)
 (inputs        a b c d e)
 (loadings      1 1 1 1 1)
 (lsi-name      . nr5)
 (out-depends   (a b c d e))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-nor5 a b c d e))))

 (gates         . 4)
 (primitives    . 1)
 (transistors   . 16))



(b-nor6
 (delays        ((146 1593) (55 807)))
 (drives        10)
 (input-types   boolp boolp boolp boolp boolp boolp)
 (inputs        a b c d e g)
 (loadings      1 1 1 1 1 1)
 (lsi-name      . nr6)
 (out-depends   (a b c d e g))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-nor6 a b c d e g))))

 (gates         . 5)
 (primitives    . 1)
 (transistors   . 18))
```

```
(b-nor8
 (delays        ((146 1853) (54 767)))
 (drives        10)
 (input-types   boolp boolp boolp boolp boolp boolp boolp boolp)
 (inputs        a b c d e g h i)
 (loadings      1 1 1 1 1 1 1 1)
 (lsi-name      . nr8)
 (out-depends   (a b c d e g h i))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-nor8 a b c d e g h i))))

 (gates         . 6)
 (primitives    . 1)
 (transistors   . 22))


(b-not
 (delays        ((70 235) (57 198)))
 (drives        10)
 (input-types   boolp)
 (inputs        a)
 (loadings      2)
 (lsi-name      . iva)
 (out-depends   (a))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-not a))))

 (gates         . 1)
 (primitives    . 1)
 (transistors   . 3))



(b-not-b4ip
 (delays        ((17 333) (12 124)))
 (drives        128)
 (input-types   boolp)
 (inputs        a)
 (loadings      8)
 (lsi-name      . b4ip)
 (out-depends   (a))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-not a))))

 (gates         . 4)
 (primitives    . 1)
 (transistors   . 12))


(b-not-ivap
 (delays        ((38 208) (35 126)))
 (drives        20)
```

```
 (input-types   boolp)
 (inputs        a)
 (loadings      3)
 (lsi-name      . ivap)
 (out-depends   (a))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-not a))))

 (gates         . 2)
 (primitives    . 1)
 (transistors   . 6))


(b-or
 (delays        ((143 332) (58 819)))
 (drives        10)
 (input-types   boolp boolp)
 (inputs        a b)
 (loadings      1 1)
 (lsi-name      . or2)
 (out-depends   (a b))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-or a b))))

 (gates         . 2)
 (primitives    . 1)
 (transistors   . 6))



(b-or3
 (delays        ((144 422) (70 1185)))
 (drives        10)
 (input-types   boolp boolp boolp)
 (inputs        a b c)
 (loadings      1 1 1)
 (lsi-name      . or3)
 (out-depends   (a b c))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-or3 a b c))))

 (gates         . 2)
 (primitives    . 1)
 (transistors   . 8))


(b-or4
 (delays        ((143 352) (78 1329)))
 (drives        10)
 (input-types   boolp boolp boolp boolp)
 (inputs        a b c d)
 (loadings      1 1 1 1)
```

```
 (lsi-name      . or4)
 (out-depends   (a b c d))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-or4 a b c d))))

 (gates         . 3)
 (primitives    . 1)
 (transistors   . 10))


(b-xor
 (delays        ((145 742) (67 973)))
 (drives        10)
 (input-types   boolp boolp)
 (inputs        a b)
 (loadings      1 2)
 (lsi-name      . eo)
 (out-depends   (a b))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-xor a b))))

 (gates         . 3)
 (primitives    . 1)
 (transistors   . 12))



(b-xor3
 (delays        ((151 1806) (79 1580)))
 (drives        10)
 (input-types   boolp boolp boolp)
 (inputs        a b c)
 (loadings      1 3 2)
 (lsi-name      . eo3)
 (out-depends   (a b c))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-xor3 a b c))))

 (gates         . 7)
 (primitives    . 1)
 (transistors   . 22))


(del2
 (delays        ((70 2035) (38 2199)))
 (drives        10)
 (input-types   boolp)
 (inputs        a)
 (loadings      1)
 (lsi-name      . del2)
 (out-depends   (a))
 (output-types  boolp)
```

```
 (outputs       z)
 (results       . ,(cons-up '((f-buf a))))

 (gates         . 4)
 (primitives    . 1)
 (transistors   . 16))


(del4
 (delays        ((73 4017) (38 4179)))
 (drives        10)
 (input-types   boolp)
 (inputs        a)
 (loadings      1)
 (lsi-name      . del4)
 (out-depends   (a))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-buf a))))

 (gates         . 7)
 (primitives    . 1)
 (transistors   . 28))



(del10
 (delays        ((60 10530) (49 10424)))
 (drives        10)
 (input-types   boolp)
 (inputs        a)
 (loadings      1)
 (lsi-name      . del10)
 (out-depends   (a))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-buf a))))

 (gates         . 19)
 (primitives    . 1)
 (transistors   . 76))




(procmon
 ;; Delay slopes are from MUX21H.  LSI flyer says delay through OR's is
 ;; at least twice the delay through the inverters, so intercepts are
 ;; approximately:    2*(30-IV's + ND2 + EN + MUX21H) ~= 25000.
 ;; (30*IV + NR2 + 90*(NR3+IVAP) + EN + MUX21H gives a much larger number.)
 (delays        ((70 25000) (60 25000)))
 (drives        10)
 (input-types   parametric parametric parametric parametric)
 (inputs        a e s n)
 (loadings      2 2 2 1)
```

```
 (lsi-name      . procmon)
 (out-depends   (a e s n))
 (output-types  parametric)
 (outputs       z)
 (results       . (cons (f-if s
                              (f-if e
                                    (f-if a '*1*false '*1*false)
                                    a)
                              n)
                        'nil))

 (gates         . 100)
 (primitives    . 1)
 (transistors   . 400))


(dp-ram-16x32
 ;; delays: Slopes are one fourth of ND2 (b-nand) slopes.
 ;;         Intercepts came from LSI flyer.
 (delays        . ,(make-list 32
                              :initial-element
                              '((35 7500) (21 7500))))
 (drives        . ,(make-list 32 :initial-element 2))
 (input-types   boolp boolp boolp boolp
                boolp boolp boolp boolp
                level
                boolp boolp boolp boolp boolp boolp boolp boolp
                boolp boolp boolp boolp boolp boolp boolp boolp
                boolp boolp boolp boolp boolp boolp boolp boolp
                boolp boolp boolp boolp boolp boolp boolp boolp)
 (inputs        ,@dp-ram-16x32-inputs)
 (loadings      2 2 2 2
                2 2 2 2
                4
                1 1 1 1 1 1 1 1
                1 1 1 1 1 1 1 1
                1 1 1 1 1 1 1 1
                1 1 1 1 1 1 1 1)
 (lsi-name      . cmrb100a)
 (new-states    . (dual-port-ram-state
                   '32 '4
                   ,(cons-up dp-ram-16x32-inputs)
                   state))
 (output-types  boolp boolp boolp boolp boolp boolp boolp boolp
                boolp boolp boolp boolp boolp boolp boolp boolp
                boolp boolp boolp boolp boolp boolp boolp boolp
                boolp boolp boolp boolp boolp boolp boolp boolp)
 (out-depends   . ,(make-list
                    32
                    :initial-element
                    '(read-a0 read-a1 read-a2 read-a3)))
 (outputs       o0  o1  o2  o3  o4  o5  o6  o7
                o8  o9  o10 o11 o12 o13 o14 o15
                o16 o17 o18 o19 o20 o21 o22 o23
                o24 o25 o26 o27 o28 o29 o30 o31)
```

```
 (results       . (dual-port-ram-value
                   '32 '4
                   ,(cons-up dp-ram-16x32-inputs)
                   state))
 (state-types   . (addressed-state 4 (ram ',(make-list 32 :initial-element
                                                        'boolp))))
 (states        . state)

 (gates         . 2368)
 (primitives    . 1)
 (transistors   . 9472))  ; Estimate: 4 * 2368



(fd1
 (delays        ((147 1024) (55 1288))
                ((145 1432) (53 1447)))
 (drives        10 10)
 (input-types   boolp clk)
 (inputs        d cp)
 (loadings      1 1)
 (lsi-name      . fd1)
 (new-states    . (f-buf d))
 (out-depends   () ())
 (output-types  boolp boolp)
 (outputs       q qn)
 (results       . ,(cons-up '((f-buf state) (f-not state))))
 (state-types   . boolp)
 (states        . state)

 (gates         . 7)
 (primitives    . 1)
 (transistors   . 26))


(fd1s
 (delays        ((147 1024) (55 1288))
                ((145 1432) (53 1447)))
 (drives        10 10)
 (input-types   boolp clk boolp boolp)
 (inputs        d cp ti te)
 (loadings      1 1 1 2)
 (lsi-name      . fd1s)
 (new-states    . (f-if te ti d))
 (out-depends   () ())
 (output-types  boolp boolp)
 (outputs       q qn)
 (results       . ,(cons-up '((f-buf state) (f-not state))))
 (state-types   . boolp)
 (states        . state)

 (gates         . 9)
 (primitives    . 1)
 (transistors   . 34))
```

```
(fd1sp
 (delays        ((68 1084) (34 1327))
                ((65 1712) (32 1596)))
 (drives        16 16)
 (input-types   boolp clk boolp boolp)
 (inputs        d cp ti te)
 (loadings      1 1 1 2)
 (lsi-name      . fd1sp)
 (new-states    . (f-if te ti d))
 (out-depends   () ())
 (output-types  boolp boolp)
 (outputs       q qn)
 (results       . ,(cons-up '((f-buf state) (f-not state))))
 (state-types   . boolp)
 (states        . state)

 (gates         . 10)
 (primitives    . 1)
 (transistors   . 38))


(fd1slp
 (delays        ((70 1085) (45  982))
                ((67 1493) (35 1568)))
 (drives        12 12)
 (input-types   boolp clk boolp boolp boolp)
 (inputs        d cp ld ti te)
 (loadings      1 1 2 1 2)
 (lsi-name      . fd1slp)
 (new-states    . (f-if te ti (f-if ld d state)))
 (out-depends   () ())
 (output-types  boolp boolp)
 (outputs       q qn)
 (results       . ,(cons-up '((f-buf state) (f-not state))))
 (state-types   . boolp)
 (states        . state)

 (gates         . 12)
 (primitives    . 1)
 (transistors   . 40))


(id
 (delays        a)
 (drives        a)
 (input-types   free)
 (inputs        a)
 (loadings      0)
 (lsi-name      . id)
 (out-depends   (a))
 (output-types  (a))
 (outputs       z)
 (results       . (cons a 'nil))
```

```
 (gates         . 0)
 (primitives    . 0)
 (transistors   . 0))


(mem-32x32
 ;; Delays are arbitrary.  Slopes are 10 times NAND slopes.
 (delays        . ,(make-list 33
                              :initial-element
                              '((1410 20000) (820 20000))))
 (drives        . ,(make-list 33 :initial-element 10))

 (input-types   boolp boolp

                boolp boolp boolp boolp boolp boolp boolp boolp
                boolp boolp boolp boolp boolp boolp boolp boolp
                boolp boolp boolp boolp boolp boolp boolp boolp
                boolp boolp boolp boolp boolp boolp boolp boolp

                boolp boolp boolp boolp boolp boolp boolp boolp
                boolp boolp boolp boolp boolp boolp boolp boolp
                boolp boolp boolp boolp boolp boolp boolp boolp
                boolp boolp boolp boolp boolp boolp boolp boolp)

 (inputs        ,@mem-32x32-inputs)

 (loadings      1 1

                1 1 1 1 1 1 1 1
                1 1 1 1 1 1 1 1
                1 1 1 1 1 1 1 1
                1 1 1 1 1 1 1 1

                1 1 1 1 1 1 1 1
                1 1 1 1 1 1 1 1
                1 1 1 1 1 1 1 1
                1 1 1 1 1 1 1 1)

 ;; The LSI-name is arbitrary.  This is a model of a memory system, NOT an
 ;; actual LSI macrocell.  The LSI-NAME is specified here because the
 ;; primitive-database predicate requires it.
 (lsi-name      . mem-32x32)

 (new-states    . (mem-state
                   ,(cons-up mem-32x32-inputs)
                   state))

 (out-depends   . ,(make-list
                    33
                    :initial-element
                    '(rw- strobe-
                      a0  a1  a2  a3  a4  a5  a6  a7
                      a8  a9  a10 a11 a12 a13 a14 a15
```

```
                     a16 a17 a18 a19 a20 a21 a22 a23
                     a24 a25 a26 a27 a28 a29 a30 a31)))

 (output-types  boolp
                tri-state tri-state tri-state tri-state
                tri-state tri-state tri-state tri-state
                tri-state tri-state tri-state tri-state
                tri-state tri-state tri-state tri-state
                tri-state tri-state tri-state tri-state
                tri-state tri-state tri-state tri-state
                tri-state tri-state tri-state tri-state
                tri-state tri-state tri-state tri-state)


 (outputs       dtack-
                o0  o1  o2  o3  o4  o5  o6  o7
                o8  o9  o10 o11 o12 o13 o14 o15
                o16 o17 o18 o19 o20 o21 o22 o23
                o24 o25 o26 o27 o28 o29 o30 o31)

 (results       . (mem-value
                   ,(cons-up mem-32x32-inputs)
                   state))

 (state-types   . ((addressed-state 32 (ram ',(make-list 32
                                                :initial-element
                                                'boolp)))
                   numberp number-listp numberp boolp boolp
                   ,(make-list 32 :initial-element 'boolp)
                   ,(make-list 32 :initial-element 'boolp)))

 (states        . state)

 (gates         . 0)
 (primitives    . 1)
 (transistors   . 0))




;; (ram-enable-circuit
;;  (clk test-regfile- disable-regfile- we)
;;  (z)
;;  ((g0 (clk-10)       del10    (clk))
;;   (g1 (test-regfile) b-not    (test-regfile-))
;;   (g2 (gate-clk)     b-or     (clk-10 test-regfile))
;;   (g3 (z)            b-nand3p (we disable-regfile- gate-clk)))
;;  nil)
;;
;; NOTE: B-NAND3P is not in the primp-database.

;; MODULE RAM-ENABLE-CIRCUIT;
;; INPUTS CLK,TEST-REGFILE-,DISABLE-REGFILE-,WE;
;; OUTPUTS Z;
;; LEVEL FUNCTION;
```

```
;; DEFINE
;; G0 (CLK-10)       = DEL10 (CLK);
;; G1 (TEST-REGFILE) = IVA   (TEST-REGFILE-);
;; G2 (GATE-CLK)     = OR2   (CLK-10,TEST-REGFILE);
;; G3 (Z)            = ND3P  (WE,DISABLE-REGFILE-,GATE-CLK);
;; END MODULE;

;; loadings:  CLK             1
;;            TEST-REGFILE-    2
;;            CLK-10          1
;;            TEST-REGFILE     1
;;            WE              2
;;            DISABLE-REGFILE- 2
;;            GATE-CLK        2
;;
;; delays:  (slope*load) + intercept + input-delay
;;  CLK-10
;;    primp delay: ((60 10530) (49 10424))
;;    low-to-high: (60 * 1) + 10530 + CLK = 10590 + CLK
;;    high-to-low: (49 * 1) + 10424 + CLK = 10473 + CLK
;;  TEST-REGFILE
;;    primp delay: ((70 235) (57 198))
;;    low-to-high: (70 * 1) + 235 + TEST-REGFILE- = 305 + TEST-REGFILE-
;;    high-to-low: (57 * 1) + 198 + TEST-REGFILE- = 255 + TEST-REGFILE-
;;  GATE-CLK
;;    primp delay: ((143 332) (58 819))
;;    low-to-high:   (143 * 2) + 332 + max(CLK-10,TEST-REGFILE)
;;                 = 618             + max(CLK-10,TEST-REGFILE)
;;    high-to-low:   ( 58 * 2) + 819 + max(CLK-10,TEST-REGFILE)
;;                 = 935             + max(CLK-10,TEST-REGFILE)
;;    range:         [618 .. 935] + max(CLK-10,TEST-REGFILE)
;;                 =        [  618 .. 935]
;;                   + max([10473..10590]+CLK, [255..305]+TEST-REGFILE-)
;;                le [11091 .. 11525] + max(CLK, TEST-REGFILE-)
;;  Z
;;    primp delay: ((68 552) (54 311))
;;    low-to-high:   (slope*load) + intercept + input-delay
;;                 = (68   * 0)   + 552       + max(WE, DISABLE-REGFILE-,
;;                                                  GATE-CLK)
;;                 le 552 + max(WE, DISABLE-REGFILE-,
;;                             [11091..11525]+max(CLK,TEST-REGFILE-))
;;                 le 552 + [11091..11525] + max(CLK, TEST-REGFILE-,
;;                                               DISABLE-REGFILE-, WE)
;;                 = [11643..12077] + max(...)
;;                 average 11860
;;    high-to-low:   (slope*load) + intercept + input-delay
;;                 = (54   * 0)   + 311       + max(WE, DISABLE-REGFILE-,
;;                                                  GATE-CLK)
;;                 le 311 + max(WE, DISABLE-REGFILE-,
;;                              [11091..11525]+max(CLK,TEST-REGFILE-))
;;                 le 311  + [11091..11525] + max(CLK, TEST-REGFILE-,
;;                                                DISABLE-REGFILE-, WE)
;;                 = [11402..11836] + max(...)
;;                 average 11619
```

```
(ram-enable-circuit
 (delays        ((68 12000) (54 12000)))
 (drives        10)
 (input-types   clk boolp boolp boolp)
 (inputs        clk test-regfile- disable-regfile- we)
 (loadings      1 2 2 2)
 (lsi-name      . ram-enable-circuit)
 (out-depends   (clk test-regfile- disable-regfile- we))
 (output-types  level)
 (outputs       z)
 (results       . ,(cons-up '((f-nand disable-regfile- we))))

 (gates         . 24)
 (primitives    . 1)
 (transistors   . 97))


(t-buf
 (delays        ((146 313) (57 728)))
 (drives        10)
 (input-types   boolp boolp)
 (inputs        e a)
 (loadings      2 2)
 (lsi-name      . (bts4 a e))       ; Note input reordering
 (out-depends   (a e))
 (output-types  tri-state)
 (outputs       z)
 (results       . ,(cons-up '((ft-buf e a))))

 (gates         . 3)
 (primitives    . 1)
 (transistors   . 12))


(t-wire
 (delays        (or  a b))
 (drives        (min a b))
 (input-types   tri-state tri-state)
 (inputs        a b)
 (loadings      1 1)
 (lsi-name      . t-wire)
 (out-depends   (a b))
 (output-types  tri-state)
 (outputs       z)
 (results       . ,(cons-up '((ft-wire a b))))

 (gates         . 0)
 (primitives    . 0)
 (transistors   . 0))


(pullup
 (delays        a)
 (drives        a)
```

```
 (input-types   tri-state)
 (inputs        a)
 (loadings      1)
 (lsi-name      . pullup)
 (out-depends   (a))
 (output-types  boolp)
 (outputs       z)
 (results       . ,(cons-up '((f-pullup a))))

 (gates         . 0)
 (primitives    . 0)
 (transistors   . 0))



;;  Delay information may not be correct for the pads.


;; Note: (1) zi delay (second one) is from LSI ttl input buffer TLCHT.
;;           (See pages 4-39 and 4-27 of LSI book.)
;;       (2) po delay (third one) has ND2 (b-nand) slopes, and
;;           intercepts are ND2 intercepts plus output ZI intercepts.
;;           (See drawings on pages 4-40 and 4-27 of LSI book.)

(ttl-bidirect
 (delays        (((61 ps-pf) 1633) ((41 ps-pf) 2253))
                (( 43         633) (20          638))
                ((141        1053) (82          799)))
 (drives        (8 mA) 10 10)
 (input-types   ttl-tri-state boolp boolp parametric)
 (inputs        io a en pi)
 (loadings      (3 pf) 1 1 1)
 (lsi-name      . bd8trp)
 (out-depends   (en a) (en a io) (en a io pi))
 (output-types  ttl-tri-state boolp parametric)
 (outputs       io zi po)
 (results       . ,(cons-up
                    '((ft-buf (f-not en) a)
                      (f-buf (ft-wire io (ft-buf (f-not en) a)))
                      (f-nand (ft-wire io (ft-buf (f-not en) a))
                              pi))))

 (gates         . 0)
 (pads          . 1)
 (primitives    . 1)
 (transistors   . 0))


(ttl-clk-input
 (delays        ((  4 1225) (  4 1152))
                ((202 1050) (117  741)))
 (drives        400 10)
 (input-types   ttl parametric)
 (inputs        a pi)
 (loadings      (3 pf) 1)
```

```
 (lsi-name      . drvt8)
 (out-depends   (a) (a pi))
 (output-types  clk parametric)
 (outputs       z po)
 (results       . ,(cons-up '((f-buf a) (f-nand a pi))))

 (gates         . 0)
 (pads          . 2)
 (primitives    . 1)
 (transistors   . 0))



;; Note: po delay (second one) has ND2 (b-nand) slopes, and
;;       intercepts are ND2 intercepts plus output Z intercepts.
;;       (See drawings on page 4-27 of LSI book.)

(ttl-input
 (delays        (( 43  633) (20 638))
                ((141 1053) (82 799)))
 (drives        10 10)
 (input-types   ttl parametric)
 (inputs        a pi)
 (loadings      (3 pf) 1)
 (lsi-name      . tlcht)
 (out-depends   (a) (a pi))
 (output-types  boolp parametric)
 (outputs       z po)
 (results       . ,(cons-up '((f-buf a) (f-nand a pi))))

 (gates         . 0)
 (pads          . 1)
 (primitives    . 1)
 (transistors   . 0))


(ttl-output
 (delays        (((61 ps-pf) 812) ((42 ps-pf) 1155)))
 (drives        (8 mA))
 (input-types   boolp)
 (inputs        a)
 (loadings      5)
 (lsi-name      . b8rp)
 (out-depends   (a))
 (output-types  ttl)
 (outputs       z)
 (results       . (cons (f-buf a) 'nil))

 (gates         . 0)
 (pads          . 1)
 (primitives    . 1)
 (transistors   . 0))


(ttl-output-parametric
```

```
 (delays        (((64 ps-pf) 737) ((42 ps-pf) 1125)))
 (drives        (4 mA))
 (input-types   parametric)
 (inputs        a)
 (loadings      3)
 (lsi-name      . b4)
 (out-depends   (a))
 (output-types  ttl)
 (outputs       z)
 (results       . (cons (f-buf a) 'nil))

 (gates         . 0)
 (pads          . 1)
 (primitives    . 1)
 (transistors   . 0))


(ttl-output-fast
 (delays        (((36 ps-pf) 991) ((24 ps-pf) 1488)))
 (drives        (8 mA))
 (input-types   boolp)
 (inputs        a)
 (loadings      3)
 (lsi-name      . b8)
 (out-depends   (a))
 (output-types  ttl)
 (outputs       z)
 (results       . (cons (f-buf a) 'nil))

 (gates         . 0)
 (pads          . 1)
 (primitives    . 1)
 (transistors   . 0))


(ttl-tri-output
 (delays        (((61 ps-pf) 1602) ((41 ps-pf) 2233)))
 (drives        (8 mA))
 (input-types   boolp boolp)
 (inputs        a en)
 (loadings      1 1)
 (lsi-name      . bt8rp)
 (out-depends   (a en))
 (output-types  ttl-tri-state)
 (outputs       z)
 (results       . (cons (ft-buf (f-not en) a) 'nil))

 (gates         . 0)
 (pads          . 1)
 (primitives    . 1)
 (transistors   . 0))


(ttl-tri-output-fast
```

```
 (delays        (((36 ps-pf) 1581) ((24 ps-pf) 2198)))
 (drives        (8 mA))
 (input-types   boolp boolp)
 (inputs        a en)
 (loadings      1 1)
 (lsi-name      . bt8)
 (out-depends   (a en))
 (output-types  ttl-tri-state)
 (outputs       z)
 (results       . (cons (ft-buf (f-not en) a) 'nil))

 (gates         . 0)
 (pads          . 1)
 (primitives    . 1)
 (transistors   . 0))



(vdd
 (delays        ((0 0) (0 0)))
 (drives        50)
 (input-types   )
 (inputs        )
 (loadings      )
 (lsi-name      . vdd)
 (out-depends   ())
 (output-types  boolp)
 (outputs       z)
 (results       . (cons '*1*true 'nil))

 (gates         . 0)
 (primitives    . 1)
 (transistors   . 0))


(vdd-parametric
 (delays        ((0 0) (0 0)))
 (drives        50)
 (input-types   )
 (inputs        )
 (loadings      )
 (lsi-name      . vdd)
 (out-depends   ())
 (output-types  parametric)
 (outputs       z)
 (results       . (cons '*1*true 'nil))

 (gates         . 0)
 (primitives    . 1)
 (transistors   . 0))


(vss
 (delays        ((0 0) (0 0)))
 (drives        50)
```

```
 (input-types   )
 (inputs        )
 (loadings      )
 (lsi-name      . vss)
 (out-depends   ())
 (output-types  boolp)
 (outputs       z)
 (results       . (cons '*1*false 'nil))

 (gates         . 0)
 (primitives    . 1)
 (transistors   . 0))

))
```

14.10 "primitives.lisp"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    PRIMITIVES.LISP
;;;
;;;  Functions and macros to automate the creation of DUAL-EVAL lemmas for the
;;;  primitives.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(in-package "USER")

;;;  Primitive result and new-state lemma macros.


(defun primitive-result (name inputs results states)
  (let* ((name& (unstring name "&"))
         (args (if inputs `(LIST ,@inputs) 'NIL))
         (state (cond
                 ((null states) 'STATE)
                 ((atom states) states)
                 (t `(LIST ,@states))))
         (value-lemma (unstring name "$VALUE"))
         (netlist (unstring name "$NETLIST")))

    `(PROGN

      (DEFN ,name& (NETLIST) T)

      (DISABLE ,name&)

      (DEFN ,netlist () NIL)

      (PROVE-LEMMA ,value-lemma (REWRITE)
        (IMPLIES
         (,name& NETLIST)
         (EQUAL (DUAL-EVAL 0 ',name ,args ,state NETLIST)
                ,results))
        ;;Hint
        ((ENABLE ,name& PRIMP DUAL-APPLY-VALUE)
         (EXPAND (DUAL-EVAL 0 ',name ,args ,state NETLIST))))

      (DISABLE ,value-lemma))))
```

```
(defun primitive-state (name inputs new-states states)
  (let* ((args (if inputs `(LIST ,@inputs) 'NIL))
         (state (cond
                 ((null states) 'STATE)
                 ((atom states) states)
                 (t `(LIST ,@states))))
         (name& (unstring name "&"))
         (state-lemma (unstring name "$STATE")))

    `(PROGN

      (PROVE-LEMMA ,state-lemma (REWRITE)
        (IMPLIES
         (,name& NETLIST)
         (EQUAL (DUAL-EVAL 2 ',name ,args ,state NETLIST)
                ,new-states))
        ;;Hint
        ((ENABLE ,name& PRIMP DUAL-APPLY-STATE)
         (EXPAND (DUAL-EVAL 2 ',name ,args ,state NETLIST))))

      (DISABLE ,state-lemma))))

;;;  COMMON-LISP-PRIMP-DATABASE is defined in "primp-database.lisp".


(defmacro result-and-state-lemmas ()
  `(PROGN
    ,@(iterate for (name . alist) in common-lisp-primp-database
        collect (primitive-result name
                                  (cdr (assoc 'inputs alist))
                                  (cdr (assoc 'results alist))
                                  (cdr (assoc 'states alist))))
    ,@(iterate for (name . alist) in common-lisp-primp-database
        when (member 'new-states alist :key #'car)
        collect (primitive-state name
                                 (cdr (assoc 'inputs alist))
                                 (cdr (assoc 'new-states alist))
                                 (cdr (assoc 'states alist))))))
```

14.11 "control.lisp"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;   CONTROL.LISP
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(in-package "USER")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    Definitions of the control states for the FM9001 control logic.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defconstant
  *control-states*
  '((fetch0  . #v00000)
    (fetch1  . #v00001)
    (fetch2  . #v00010)
    (fetch3  . #v00011)

    (decode  . #v00100)
    (rega    . #v00101)
    (regb    . #v00110)
    (update  . #v00111)

    (reada0  . #v01000)
    (reada1  . #v01001)
    (reada2  . #v01010)
    (reada3  . #v01011)

    (readb0  . #v01100)
    (readb1  . #v01101)
    (readb2  . #v01110)
    (readb3  . #v01111)

    (write0  . #v10000)
    (write1  . #v10001)
    (write2  . #v10010)
    (write3  . #v10011)

    (sefa0   . #v10100)
    (sefa1   . #v10101)
    (sefb0   . #v10110)
    (sefb1   . #v10111)

    (hold0   . #v11000)
    (hold1   . #v11001)
    (v11010  . #v11010)
    (v11011  . #v11011)

    (reset0  . #v11100)
    (reset1  . #v11101)
    (reset2  . #v11110)
    (v11111  . #v11111)))

;;;  Define both vector (V_) and natural (N_) forms of the states, disable them
;;;  and their *1* counterparts, and define a theory for these events.


(defun vector-state-name (state) (unstring "V_" state))

(defun vector-state-name*1* (state) (unstring "*1*V_" state))

(defun natural-state-name (state) (unstring "N_" state))

(defun natural-state-name*1* (state) (unstring "*1*N_" state))
```

```
(defmacro define-control-states ()
  `(PROGN

    ,@(iterate for (name . val) in *control-states*
        nconc (let* ((vn (vector-state-name name))
                     (vn*1* (vector-state-name*1* name))
                     (nn (natural-state-name name))
                     (nn*1* (natural-state-name*1* name)))
               `((DEFN ,vn () ,val)
                 (DISABLE ,vn)
                 (DISABLE ,vn*1*)
                 (DEFN ,nn () (V-TO-NAT ,val))
                 (DISABLE ,nn)
                 (DISABLE ,nn*1*))))

    (DEFTHEORY VECTOR-STATE-THEORY
      ,(iterate for (name . val) in *control-states*
         collecting (vector-state-name name)))

    (DEFTHEORY NATURAL-STATE-THEORY
      ,(iterate for (name . val) in *control-states*
         collecting (natural-state-name name)))

    (PROVE-LEMMA BVP-LENGTH-STATE-VECTORS (REWRITE)
     (AND ,@(iterate for (name . val) in *control-states*
          nconc `((EQUAL (LENGTH (,(vector-state-name name))) 5)
                  (BVP (,(vector-state-name name))))))
     ;;Hint
     ((ENABLE-THEORY VECTOR-STATE-THEORY)))

    (DISABLE BVP-LENGTH-STATE-VECTORS)

    ))


;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;   *CONTROL-TEMPLATE*
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

;;;  The fields of the control state are defined by *CONTROL-TEMPLATE*.  We
;;;  make extensive use of this database to generate lemmas.
```

```
(defconstant
  *control-template*
  ;; Line       Type (Bit/Vector)  Default
  '((rw-                b          t)
    (strobe-            b          t)
    (hdack-             b          t)
    (we-regs            b          f)
    (we-a-reg           b          f)
    (we-b-reg           b          f)
    (we-i-reg           b          f)
    (we-data-out        b          f)
    (we-addr-out        b          f)
    (we-hold-           b          f)
    (we-pc-reg          b          f)
    (data-in-select     b          f)
    (dec-addr-out       b          f)
    (select-immediate   b          f)
    (alu-c              b          (carry-in-help (cons c (cons f op-code))))
    (alu-zero           b          f)
    (state              5          #v00000)
    (we-flags           4          (make-list 4 f))
    (regs-address       4          pc-reg)
    (alu-op             4          op-code)
    (alu-mpg            7          (mpg (cons f (cons f op-code))))))

;;;  Define an accessor for each entry in the control state, and a theory for
;;;  all of the accessors.
```

```
(defmacro define-control-state-accessors ()
  `(PROGN

    ,@(iterate for (field type default) in *control-template*
        with pos = 0
        nconc (if (equal type 'b)
                  (prog1
                    `((DEFN ,field (CNTL-STATE) (NTH ,pos CNTL-STATE))
                      (DISABLE ,field))
                    (incf pos))
                (let ((size type))
                  (prog1
                    `((DEFN ,field (CNTL-STATE)
                        (SUBRANGE CNTL-STATE ,pos ,(1- (+ pos size))))
                      (DISABLE ,field))
                    ;; Hack to work around variable unused error message.
                    (ignore-variable default)
                    (setf pos (+ pos size))))))

    (DEFTHEORY CONTROL-STATE-ACCESSOR-THEORY
      ,(mapcar #'car *control-template*))))



;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    CONTROL-LET
;;;
;;;  A macro for a LET that extracts and computes necessary fields and flags.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defconstant *control-arglist*
  '(RW- REGS-ADDRESS I-REG FLAGS PC-REG))
```

```
(defun control-let (body)
  `(LET ((A-IMMEDIATE-P (A-IMMEDIATE-P I-REG))
         (MODE-A        (MODE-A        I-REG))
         (RN-A          (RN-A          I-REG))
         (MODE-B        (MODE-B        I-REG))
         (RN-B          (RN-B          I-REG))
         (SET-FLAGS     (SET-FLAGS     I-REG))
         (STORE-CC      (STORE-CC      I-REG))
         (OP-CODE       (OP-CODE       I-REG)))
     (LET ((A-IMMEDIATE-P-     (NOT* A-IMMEDIATE-P))
           (STORE              (STORE-RESULTP STORE-CC FLAGS))
           (SET-SOME-FLAGS     (SET-SOME-FLAGS SET-FLAGS))
           (DIRECT-A           (OR* A-IMMEDIATE-P (REG-DIRECT-P MODE-A)))
           (DIRECT-B           (REG-DIRECT-P MODE-B))
           (UNARY              (UNARY-OP-CODE-P OP-CODE))
           (PRE-DEC-A          (PRE-DEC-P MODE-A))
           (POST-INC-A         (POST-INC-P MODE-A))
           (PRE-DEC-B          (PRE-DEC-P MODE-B))
           (POST-INC-B         (POST-INC-P MODE-B))
           (C                  (C-FLAG FLAGS))
           (ALL-T-REGS-ADDRESS (EQUAL REGS-ADDRESS #v1111)))
       (LET ((STORE-        (NOT* STORE))
             (DIRECT-A-     (NOT* DIRECT-A))
             (DIRECT-B-     (NOT* DIRECT-B))
             (UNARY-        (NOT* UNARY))
             (SIDE-EFFECT-A (AND* A-IMMEDIATE-P-
                                  (OR* PRE-DEC-A POST-INC-A)))
             (SIDE-EFFECT-B (OR* PRE-DEC-B POST-INC-B)))
         ,body))))



;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    *STATE-TABLE*
;;;
;;;    *STATE-TABLE*, along with *CONTROL-TEMPLATE*, forms a Common LISP
;;;    encoding of the FM9001 state machine. Each entry contains the
;;;    state name, an expression for the next state, and a set of control line
;;;    assignments.  If the name of a control line is given, then the
;;;    non-default Boolean value is selected.  A pair, (<control-line> <value>)
;;;    assigns that value to the control line in that state.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defconstant
  *state-table*
  '((fetch0 fetch1 we-addr-out we-a-reg (regs-address pc-reg)
            (rw- rw-) we-hold-
            alu-zero
            (alu-op (alu-inc-op))
            (alu-c (carry-in-help (cons c (cons t (alu-inc-op)))))
            (alu-mpg (mpg (cons t (cons f (alu-inc-op))))))

    (fetch1 (if hold- fetch2 hold0))

    (fetch2 fetch3 strobe-
            we-regs (regs-address pc-reg)
            (alu-c (carry-in-help (cons c (cons f (alu-inc-op)))))
            (alu-op (alu-inc-op))
            (alu-mpg (mpg (cons f (cons f (alu-inc-op))))))

    (fetch3 (if dtack- fetch3 decode) data-in-select we-i-reg strobe-)

    (decode (if (b-or store set-some-flags)
                (if direct-a
                    (if (b-or direct-b unary)
                        rega
                      readb0)
                  reada0)
              (if side-effect-a
                  sefa0
                (if side-effect-b
                    sefb0
                  fetch0))))

    (rega (if direct-b (if unary update regb) (if store write0 update))
          (regs-address rn-a) (select-immediate a-immediate-p) we-a-reg)

    (regb update (regs-address rn-b) we-b-reg)

    (update (if (b-and side-effect-b unary) sefb1 fetch0)
            (regs-address rn-b) (we-regs store) (we-flags set-flags)
            we-b-reg)

    (reada0 reada1 (regs-address rn-a) we-a-reg we-addr-out
            (dec-addr-out pre-dec-a))

    (reada1 reada2
            (alu-c (if* pre-dec-a
                        (carry-in-help (cons c (cons f (alu-dec-op))))
                      (carry-in-help (cons c (cons f (alu-inc-op))))))
            (alu-op (if* pre-dec-a (alu-dec-op) (alu-inc-op)))
            (alu-mpg (if* pre-dec-a
                          (mpg (cons f (cons f (alu-dec-op))))
                        (mpg (cons f (cons f (alu-inc-op))))))
            (regs-address rn-a) (we-regs side-effect-a))

    (reada2 reada3 (regs-address rn-b) we-b-reg strobe-)
```

```
(reada3 (if dtack-
            reada3
          (if direct-b
              update
            (if unary
                (if store write0 update)
              readb0)))
        data-in-select we-a-reg strobe-)

(readb0 readb1 (regs-address rn-b) we-addr-out
        (dec-addr-out pre-dec-b) we-b-reg)

(readb1 readb2 (regs-address rn-a) (we-a-reg direct-a)
        (select-immediate a-immediate-p))

(readb2 readb3
        (regs-address rn-b) (we-regs (and* store- side-effect-b))
        (alu-c (if* pre-dec-b
                    (carry-in-help (cons c (cons f (alu-dec-op))))
                  (carry-in-help (cons c (cons f (alu-inc-op))))))
        (alu-op (if* pre-dec-b (alu-dec-op) (alu-inc-op)))
        (alu-mpg (if* pre-dec-b
                      (mpg (cons f (cons t (alu-dec-op))))
                    (mpg (cons f (cons t (alu-inc-op))))))
        strobe-)

(readb3 (if dtack- readb3 (if store write0 update))
        we-b-reg data-in-select strobe-)

(write0 write1
        (we-flags set-flags) we-data-out
        (regs-address rn-b) we-b-reg we-addr-out
        (dec-addr-out pre-dec-b))

(write1 write2
        (regs-address rn-b) (we-regs side-effect-b)
        (alu-c (if* pre-dec-b
                    (carry-in-help (cons c (cons f (alu-dec-op))))
                  (carry-in-help (cons c (cons f (alu-inc-op))))))
        (alu-op (if* pre-dec-b (alu-dec-op) (alu-inc-op)))
        (alu-mpg (if* pre-dec-b
                      (mpg (cons f (cons t (alu-dec-op))))
                    (mpg (cons f (cons t (alu-inc-op))))))
        rw-)

(write2 write3 strobe- rw-)

(write3 (if dtack- write3 fetch0) rw- strobe-)

(sefa0 sefa1 (regs-address rn-a) we-a-reg)

(sefa1 (if side-effect-b sefb0 fetch0)
       (regs-address rn-a) we-regs
       (alu-c (if* pre-dec-a
```

```
                 (carry-in-help (cons c (cons f (alu-dec-op))))
               (carry-in-help (cons c (cons f (alu-inc-op))))))
       (alu-op (if* pre-dec-a (alu-dec-op) (alu-inc-op)))
       (alu-mpg (if* pre-dec-a
                   (mpg (cons f (cons f (alu-dec-op))))
                 (mpg (cons f (cons f (alu-inc-op)))))))


 (sefb0 sefb1 (regs-address rn-b) we-b-reg)

 (sefb1 fetch0
       (regs-address rn-b) we-regs
       (alu-c (if* pre-dec-b
                 (carry-in-help (cons c (cons f (alu-dec-op))))
               (carry-in-help (cons c (cons f (alu-inc-op))))))
       (alu-op (if* pre-dec-b (alu-dec-op) (alu-inc-op)))
       (alu-mpg (if* pre-dec-b
                   (mpg (cons f (cons t (alu-dec-op))))
                 (mpg (cons f (cons t (alu-inc-op)))))))

 (hold0 (if hold- hold1 hold0) hdack- we-pc-reg we-hold-)

 (hold1 fetch0)

 (v11010 reset0)                    ;Illegal
 (v11011 reset0)                    ;Illegal

 ;;  We use (ALU-INC-OP) as the default op-code during the reset sequence
 ;;  so that the control vector will be completely defined.  Recall that the
 ;;  OP-CODE is irrelevant to ALU operation when it is forced to zero.
 ;;  Thu Jul 18 11:33:55 1991 BB -- With new optimizations, we depend on the
 ;;  OP-CODE being set to ALU-INC-OP during zeroing!
 ;;  NB: All idiomatic expressions are necessary for later proofs; cryptic
 ;;  comments being especially opaque.

 (reset0 reset1
        (regs-address (make-list 4 f)) we-regs we-data-out
        (we-flags (make-list 4 t)) we-pc-reg we-hold-
        alu-zero
        (alu-op (alu-inc-op))
        (alu-c (carry-in-help (cons c (cons t (alu-inc-op)))))
        (alu-mpg (mpg (cons t (cons f (alu-inc-op))))))

 (reset1 reset2 (regs-address (make-list 4 f))
        we-addr-out we-a-reg we-b-reg we-i-reg
        alu-zero
        (alu-op (alu-inc-op))
        (alu-c (carry-in-help (cons c (cons t (alu-inc-op)))))
        (alu-mpg (mpg (cons t (cons f (alu-inc-op))))))

 (reset2 (if all-t-regs-address fetch0 reset2)
        we-regs
        alu-zero
        (alu-op (alu-inc-op))
        (alu-c (carry-in-help (cons c (cons t (alu-inc-op)))))
```

```
          (alu-mpg (mpg (cons t (cons f (alu-inc-op)))))
          (regs-address (v-inc regs-address)))

   (v11111 reset0)))                  ;Illegal



;;;  For each control state, define a control vector function, and lemmas to
;;;  destructure the control vector function.


(defun cv-name (state) (unstring "CV_" state))

(defun cv-lemma-name (state) (unstring "CV_" state "$DESTRUCTURE"))

;;;   Use the *STATE-TABLE* to build a CV_state function for each state.  This
;;;   is the function that creates the control-vector for each state.
;;;   Note that the reset states RESET0 and RESET1 are constants, and
;;;   in these cases we don't include the hypothesis.
```

```
(defmacro define-control-vector-functions ()
  (labels
    ((build-state
      (template)
      (cond
       ((null template) 'NIL)
       (t (if (numberp (cadar template))
              `(APPEND ,(caddar template) ,(build-state (cdr template)))
            `(CONS ,(caddar template) ,(build-state (cdr template))))))))

    `(PROGN
      ,@(iterate for (state next-state . fields) in *state-table*
          nconc (progn (ignore-variable next-state)
                       (let ((template (copy-tree *control-template*)))
                         (setf (third (assoc 'state template))
                               (list (vector-state-name state)))
                         (dolist (field fields)
                           (let ((triple (assoc (if (listp field)
                                                    (car field)
                                                  field)
                                                template)))
                             (unless triple
                               (error "No field named ==> ~s." field))
                             (setf (caddr triple)
                                   (if (listp field)
                                       (cadr field)
                                     (if (equal (caddr triple) 'T)
                                         'F
                                       'T)))))
                         `((DEFN
                             ,(cv-name state)
                             ,*control-arglist*
                             ,(control-let (build-state template)))
                           (DISABLE ,(cv-name state))
                           (#+axioms
                            ADD-AXIOM
                            #-axioms
                            PROVE-LEMMA
                            ,(cv-lemma-name state) (REWRITE)
                            ,(control-let
                              `(IMPLIES
                                ,(if (member state '(RESET0 RESET1))
                                     'T
                                   '(CV-HYPS RW- REGS-ADDRESS I-REG
                                             FLAGS PC-REG))
                                (AND
                                 ,@(iterate for (field type value) in template
                                     collect
                                     (progn
                                       (ignore-variable type)
                                       `(EQUAL (,field (,(cv-name state)
                                                        ,@*control-arglist*))
                                               ,value))))))
```

```
                          ;;Hint
                          #-axioms
                          ((ENABLE ,(cv-name state)
                                   ,(vector-state-name state)
                                   OPEN-NTH NTH-IF
                                   SUBRANGE-CONS OP-CODE SET-FLAGS RN-A RN-B)
                           (ENABLE-THEORY CONTROL-STATE-ACCESSOR-THEORY)
                           (DISABLE MPG)))
                         (DISABLE ,(cv-lemma-name state))))))

     (DEFTHEORY CV-THEORY
       ,(iterate for (state . rest) in *state-table*
          collect (progn (ignore-variable rest)
                         (cv-name state))))

     (DEFTHEORY CV-DESTRUCTURE-THEORY
       ,(iterate for (state . rest) in *state-table*
          collect (progn (ignore-variable rest)
                         (cv-lemma-name state))))

     (PROVE-LEMMA BVP-CV-VECTORS (REWRITE)
       (IMPLIES
        (CV-HYPS RW- REGS-ADDRESS I-REG FLAGS PC-REG)
        (AND ,@(iterate for (state . rest) in *state-table*
                 collect
                 (progn (ignore-variable rest)
                        `(BVP (,(cv-name state)
                               RW- REGS-ADDRESS I-REG FLAGS PC-REG))))))
       ;;Hint
       ((ENABLE-THEORY CV-THEORY)
        (ENABLE BVP-LENGTH-STATE-VECTORS)))

     (DISABLE BVP-CV-VECTORS))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    SYNTHESIS
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;    Symbolic gate generation.
```

```
(defun b-not-ify (args)
  (let (arg
        (alist
         `((B-OR . B-NOR)
           (B-OR3 . B-NOR3)
           (B-OR4 . B-NOR4)
           (B-OR5 . B-NOR5)
           (B-OR6 . B-NOR6)
           (B-OR8 . B-NOR8)
           (B-AND . B-NAND)
           (B-AND3 . B-NAND3)
           (B-AND4 . B-NAND4)
           (B-AND5 . B-NAND5)
           (B-AND6 . B-NAND6)
           (B-AND8 . B-NAND8))))
    (if (match (car args) (VSS))
        '(VDD)
      (if (match (car args) (VDD))
          '(VSS)
        (if (match (car args) (B-NOT arg))
            arg
          (if (and (listp (car args)) (assoc (caar args) alist))
              (cons (cdr (assoc (caar args) alist))
                    (cdar args))
            (if (and (listp (car args)) (rassoc (caar args) alist))
                (cons (car (rassoc (caar args) alist))
                      (cdar args))
              `(B-NOT ,(car args)))))))))


(defun b-nand-ify (args)
  (let ((l (length args)))
    (case l
      (0 '(VSS))
      (1 (b-not-ify args))
      (2 `(B-NAND ,@args))
      (7 `(B-NAND (B-AND4 ,@(subseq args 0 4))
                  (B-AND3 ,@(nthcdr 4 args))))
      (t `(,(unstring "B-NAND" l) ,@args)))))


(defun b-and-ify (args)
  (let ((l (length args)))
    (case l
      (0 '(VDD))
      (1 (car args))
      (2 `(B-AND ,@args))
       (7 `(B-NOR (B-NAND4 ,@(subseq args 0 4))
                 (B-NAND3 ,@(nthcdr 4 args))))
      (t `(,(unstring "B-AND" l) ,@args)))))
```

```
(defun b-or-ify (args)
  (let ((l (length args)))
    (case l
      (0 '(VSS))
      (1 (car args))
      (2 `(B-OR ,@args))
      (7 `(B-NAND (B-NOR4 ,@(subseq args 0 4))
                  (B-NOR3 ,@(nthcdr 4 args))))
      (t `(,(unstring "B-OR" l) ,@args)))))


(defun b-nor-ify (args)
  (let ((l (length args)))
    (case l
      (0 '(VSS))
      (1 (b-not-ify args))
      (2 `(B-NOR ,@args))
      (7 `(B-NOR (B-OR4 ,@(subseq args 0 4))
                 (B-OR3 ,@(nthcdr 4 args))))
      (t `(,(unstring "B-NOR" l) ,@args)))))

;;;  This macro defines NEXT-STATE*, which takes the current decoded state
;;;  and creates a decoded version of the next state.
```

```
(defmacro define-next-state ()
  (let ((state-names (mapcar #'car *control-states*))
        (state-alist (iterate for (state . val) in *control-states*
                       collecting (list state)))
        spec hdl-body)

    (labels

      ((sname (state) (unstring "S" (position state state-names)))

       (siname (state) `(INDEX 'S ,(position state state-names)))

       (unwind
        (tree expr)
        (cond
         ((symbolp tree) (push expr (cdr (assoc tree state-alist))))
         ((equal (car tree) 'IF)
          (unwind (caddr tree) (cons (cadr tree) expr))
          (unwind (cadddr tree) (cons `(B-NOT ,(cadr tree)) expr)))
         (t (error "DEFINE-NEXT-STATE error"))))

       (cons-up
        (alist)
        (cond
         ((null alist) ''nil)
         (t `(CONS ,(b-nand-ify (iterate for clause in (cdar alist)
                                 collecting (b-nand-ify clause)))
                   ,(cons-up (cdr alist)))))))

      (let (hdl-outputs
            ;;  (snames (mapcar #'sname state-names))
            ;;  (sinames (mapcar #'siname state-names))
            )

        (iterate for (state next . rest) in *state-table*
          do (progn (ignore-variable rest)
                    (unwind next (list state))))

        (setf spec (cons-up state-alist))

        (multiple-value-setq (hdl-outputs hdl-body) (logic-to-hdl spec))

        `(PROGN

          (DEFN NEXT-STATE (DECODED-STATE
                            STORE SET-SOME-FLAGS
                            UNARY DIRECT-A DIRECT-B
                            SIDE-EFFECT-A SIDE-EFFECT-B
                            ALL-T-REGS-ADDRESS
                            DTACK- HOLD-)
            (LET ,(iterate for name in state-names for n from 0
                    collecting `(,name (NTH ,n DECODED-STATE)))
              ,spec))
```

```
(DEFN F$NEXT-STATE (DECODED-STATE
                    STORE SET-SOME-FLAGS
                    UNARY DIRECT-A DIRECT-B
                    SIDE-EFFECT-A SIDE-EFFECT-B
                    ALL-T-REGS-ADDRESS
                    DTACK- HOLD-)
  (LET ,(iterate for name in state-names for n from 0
          collecting `(,name (NTH ,n DECODED-STATE)))
    ,(f-body spec)))

(DISABLE F$NEXT-STATE)

(PROVE-LEMMA F$NEXT-STATE=NEXT-STATE (REWRITE)
  (IMPLIES
   (AND (BVP DECODED-STATE)
        (EQUAL (LENGTH DECODED-STATE) 32)
        (BOOLP STORE) (BOOLP SET-SOME-FLAGS)
        (BOOLP UNARY) (BOOLP DIRECT-A) (BOOLP DIRECT-B)
        (BOOLP SIDE-EFFECT-A) (BOOLP SIDE-EFFECT-B)
        (BOOLP ALL-T-REGS-ADDRESS)
        (BOOLP DTACK-) (BOOLP HOLD-))
   (EQUAL (F$NEXT-STATE DECODED-STATE
                    STORE SET-SOME-FLAGS
                    UNARY DIRECT-A DIRECT-B
                    SIDE-EFFECT-A SIDE-EFFECT-B
                    ALL-T-REGS-ADDRESS
                    DTACK- HOLD-)
          (NEXT-STATE DECODED-STATE
                    STORE SET-SOME-FLAGS
                    UNARY DIRECT-A DIRECT-B
                    SIDE-EFFECT-A SIDE-EFFECT-B
                    ALL-T-REGS-ADDRESS
                    DTACK- HOLD-)))
  ;;HINT
  ((ENABLE F$NEXT-STATE NEXT-STATE BOOLP-B-GATES)
   (DISABLE-THEORY F-GATES B-GATES)))

(DEFN NEXT-STATE* ()
  (LIST
   'NEXT-STATE
   (APPEND #i(s 0 32)
           '(STORE SET-SOME-FLAGS
                   UNARY DIRECT-A DIRECT-B
                   SIDE-EFFECT-A SIDE-EFFECT-B
                   ALL-T-REGS-ADDRESS
                   DTACK- HOLD-))
   ',hdl-outputs
   (APPEND
    (LIST
     ,@(iterate for state in state-names for n from 0
        collecting
        `(LIST ',(unstring "R-" n) '(,state) 'ID (LIST #i(S ,n)))))
    ',hdl-body)
   nil))
```

```
          (MODULE-PREDICATE NEXT-STATE*)

          (MODULE-NETLIST NEXT-STATE*))))))

#|
This stuff that is commented out was used to help build the specification and
DUAL-EVAL form of the control logic.


(defun partition-set (fn set)
  (iterate for el in set with p with key
    do (let* ((key (funcall fn el))
              (pair (assoc-equal key p)))
         (if pair
             (push el (cdr pair))
           (setf p (acons key (list el) p))))
    finally (return p)))


(setf equations
      (iterate for (field type default) in (cdr *control-template*)
        collecting
        (let (assignments)
          (iterate for (state next . fields) in *state-table*
            do (iterate for assignment in fields
                 do (if (equal assignment field)
                        (if (equal type 'b)
                            (if (equal default 'T)
                                (push (cons state 'F) assignments)
                              (push (cons state 'T) assignments))
                          (error "Wrong type ~s ~s" assignment type))
                      (if (consp assignment)
                          (if (equal (car assignment) field)
                              (unless (equal (cadr assignment) default)
                                (push (cons state (cadr assignment))
                                      assignments)))))))
          (list field type default
                (iterate for (key . pairs) in (partition-set #'cdr assignments)
                  collecting (cons key (mapcar #'car pairs)))))))
```

```
(iterate for (field type default assignments) in equations
  collecting
  (list field type default
        (if (and (equal type 'b) (not (equal field 'ALU-C)))
            (if (equal default 'f)
                (b-nand-ify
                 (iterate for (value . disjuncts) in assignments
                   collecting
                   (if (equal value 't)
                       (b-nor-ify disjuncts)
                     `(B-NAND ,value ,(b-or-ify disjuncts)))))
              (b-and-ify
               (iterate for (value . disjuncts) in assignments
                 collecting
                 (if (equal value 'f)
                     (b-nor-ify disjuncts)
                   `(B-NAND (B-NOT ,value) ,(b-or-ify disjuncts))))))
          assignments)))

|#



;;;  Write a lemma for the next control-state for each state in terms of the CV
;;;  functions.
```

```
(defmacro generate-next-cntl-state-lemmas ()
  (labels
    ((translate-b-fns
      (form)
      (cond
       ((symbolp form) form)
       (t (case (car form)
            (b-and (cons 'AND* (mapcar #'translate-b-fns (cdr form))))
            (b-or  (cons 'OR*  (mapcar #'translate-b-fns (cdr form))))
            (b-not (cons 'NOT* (mapcar #'translate-b-fns (cdr form))))
            (t (error "Bad form ==> ~s." form))))))

     (make-if-tree
      (tree)
      (cond
       ((symbolp tree) `(,(cv-name tree) ,@*control-arglist*))
       ((and (consp tree) (equal (car tree) 'IF))
        `(IF* ,(translate-b-fns (cadr tree))
              ,(make-if-tree (caddr tree))
            ,(make-if-tree (cadddr tree))))
       (t (error "Bad tree => ~s." tree)))))

    `(PROGN
      ,@(iterate for (state next-state . fields) in *state-table*
          collecting
          (progn
            (ignore-variable fields)
            (let ((v-state `(,(vector-state-name state))))
              `(PROVE-LEMMA ,(unstring "NEXT-CNTL-STATE$" state) (REWRITE)
                 (IMPLIES
                  (AND (EQUAL RESET- T)
                       (CV-HYPS RW- REGS-ADDRESS I-REG FLAGS PC-REG))
                  (EQUAL (NEXT-CNTL-STATE RESET- DTACK- HOLD- RW- ,v-state
                                          I-REG FLAGS PC-REG REGS-ADDRESS)
                         ,(control-let (make-if-tree next-state))))
                 ;;Hint
                 ((ENABLE-THEORY VECTOR-STATE-THEORY
                                 CV-THEORY IR-FIELDS-THEORY)
                  (ENABLE CV NEXT-CNTL-STATE IF*)
                  (DISABLE *1*CARRY-IN-HELP *1*MPG)))))))))
```

14.12 "expand-fm9001.lisp"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    EXPAND-FM9001.LISP
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(in-package "USER")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    SINGLE STEP SIMULATIONS
;;;
;;;  The STEP-FM9001 macro creates a lemma that describes the new state
;;;  computed by a single step of the low-level machine from the given initial
;;;  state.  This procedure is complicated by the fact that the next state
;;;  depends on the state of the memory.  Rather than produce a single lemma
;;;  that captures the behavior for all possible memory states, we create
;;;  separate lemmas only for those processor-state/memory-state configurations
;;;  that are possible during normal instruction execution.  This is important
;;;  because if the memory protocol is not followed, then unknown values may be
;;;  loaded into the processor registers.  What we are doing here is creating
;;;  a state-by-state, almost-Boolean specification of the low-level machine.
;;;  So we obviously we want to insure that the processor state remains
;;;  Boolean.
;;;
;;;  STEP-FM9001 is used in "expand-fm9001.events".
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defmacro step-fm9001 (state &key
                             (suffix "")
                             (addr-stable nil)
                             (data-stable nil)
                             (dtack- 'DTACK-)
                             (mem-state 0)
                             (mem-count 'MEM-COUNT)
                             (mem-dtack 'MEM-DTACK)
                             (last-rw- 'T)
                             (hyps nil)
                             (enable nil)
                             (disable nil)
                             (split-term nil))
  (let ((term
         `(RUN-FM9001
           (LIST
            (LIST REGS
                  FLAGS
                  A-REG B-REG I-REG
                  DATA-OUT ADDR-OUT
                  RESET- ,dtack- HOLD-
                  PC-REG
                  (,(cv-name state)
                   ,last-rw- LAST-REGS-ADDRESS LAST-I-REG
                   LAST-FLAGS LAST-PC-REG))
            (LIST MEM ,mem-state CLOCK ,mem-count ,mem-dtack
                  ,last-rw-
                  ,(if addr-stable 'ADDR-OUT 'LAST-ADDRESS)
                  ,(if data-stable 'DATA-OUT 'LAST-DATA)))
           INPUTS
           N))
        (hyps
         `(AND (CV-HYPS ,last-rw- LAST-REGS-ADDRESS LAST-I-REG LAST-FLAGS
                        LAST-PC-REG)
               (CV-HYPS T PC-REG I-REG FLAGS PC-REG)
               (MEMORY-OKP 32 32 MEM)
               (regfile-okp regs)
               (BVP A-REG)
               (EQUAL (LENGTH A-REG) 32)
               (BVP B-REG)
               (EQUAL (LENGTH B-REG) 32)
               (BVP ADDR-OUT)
               (EQUAL (LENGTH ADDR-OUT) 32)
               (BVP DATA-OUT)
               (EQUAL (LENGTH DATA-OUT) 32)
               (NOT (ZEROP N))
               (RUN-INPUTS-P INPUTS 1)
               (boolp ,dtack-)
               (boolp HOLD-)
               (EQUAL RESET- T)
               ,hyps))
        (hints
         `((ENABLE-THEORY FM9001-HARDWARE-STATE-ACCESSORS)
           (DISABLE-THEORY f-gates)
```

```
          (ENABLE RUN-FM9001-STEP-CASE
                  rewrite-FM9001-NEXT-STATE-for-step-lemmas boolp-b-gates
                  NEXT-MEMORY-STATE MEMORY-VALUE
                  ,(cv-lemma-name state) OPEN-NTH
                  bvp-length-state-vectors bvp-cv-vectors
                  ,@enable)
          (DISABLE MPG MAKE-TREE *1*MAKE-TREE *1*make-list
                   open-v-threefix ,@disable))))

    `(#+axioms
      ADD-AXIOM
      #-axioms
      PROVE-LEMMA
      ,(unstring state "$STEP" suffix) (REWRITE)
      (IMPLIES
       ,hyps
       (EQUAL ,term
              ,(if split-term
                   `(IF ,split-term
                        ,(print (expand term `(AND ,hyps ,split-term) hints))
                      ,(print (expand term `(AND ,hyps (NOT ,split-term))
                                      hints)))
                 (print (expand term hyps hints)))))
      ;;Hint
      #-axioms
      ,hints
      )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    Timing functions
;;;
;;;  T_initial-state->final-state is an expression for the amount of time
;;;  necessary to execute the indicated section of the state diagram.
;;;  CT_initial-state->final-state is an expression for the memory delay oracle
;;;  at the end of the sequence.  TIMING-IF-TREE creates an "IF" tree of calls
;;;  to timing functions based on the branching decisions that appear in the
;;;  state diagrams.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defun t_fetch0->fetch0 ()
  1)


(defun ct_fetch0->fetch0 ()
  'CLOCK)


(defun w_fetch1->decode ()
  '(CAR CLOCK))
```

```
(defun t_fetch1->decode ()
  '(ADD1 (ADD1 (PLUS #.(w_fetch1->decode) 2))))

(defun ct_fetch1->decode ()
  '(CDR CLOCK))

(defun t_rega->rega ()
  1)

(defun ct_rega->rega ()
  'CLOCK)

(defun t_regb->update ()
  2)

(defun ct_regb->update ()
  'CLOCK)

(defun t_update->update ()
  1)

(defun ct_update->update ()
  'CLOCK)

(defun w_reada0->reada3 ()
  '(CAR CLOCK))

(defun t_reada0->reada3 ()
  '(ADD1 (ADD1 (ADD1 (PLUS #.(w_reada0->reada3) 1)))))

(defun ct_reada0->reada3 ()
  '(CDR CLOCK))

(defun w_readb0->readb3 ()
  '(CAR CLOCK))

(defun t_readb0->readb3 ()
  '(ADD1 (ADD1 (ADD1 (PLUS #.(w_readb0->readb3) 1)))))
```

```
(defun ct_readb0->readb3 ()
  '(CDR CLOCK))


(defun w_write0->write3 ()
  '(CAR CLOCK))


(defun t_write0->write3 ()
  '(ADD1 (ADD1 (ADD1 (PLUS #.(w_write0->write3) 1)))))


(defun ct_write0->write3 ()
  '(CDR CLOCK))


(defun t_sefa0->sefa1 ()
  2)


(defun ct_sefa0->sefa1 ()
  'CLOCK)


(defun t_sefb0->sefb1 ()
  2)


(defun ct_sefb0->sefb1 ()
  'CLOCK)


(defun t_sefb1->sefb1 ()
  1)


(defun ct_sefb1->sefb1 ()
  'CLOCK)
```

```
(defun timing-if-tree (state)
  (labels
    ((timify-tree
      (tree)
      (cond
       ((symbolp tree) `(,(unstring "T_" tree) CLOCK I-REG FLAGS))
       ((and (consp tree) (equal (car tree) 'IF))
        (if (equal (cadr tree) 'DTACK-)
            (timify-tree (cadddr tree))
          (if (equal (cadr tree) 'HOLD-)
              (timify-tree (caddr tree))
            `(IF ,(cadr tree)
                 ,(timify-tree (caddr tree))
               ,(timify-tree (cadddr tree))))))
       (t (error "Bad tree => ~s." tree)))))
    (control-let (timify-tree (cadr (assoc state *state-table*))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    Chunk Simulations
;;;
;;;  SIM-FM9001 is a macro that creates lemmas that describe the operation of
;;;  the machine for multiple clock cycles.  We create a multi-state simulation
;;;  for each non-branching segment of the state diagrams.  For this purpose,
;;;  we consider the states that wait for memory as "non-branching".  Notice
;;;  that some of the "multi-state" lemmas only include a single state.
;;;
;;;  SIM-FM9001 is used in "expand-fm9001.events"
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defmacro sim-fm9001 (start-state end-state
                                  &key
                                  (suffix "")
                                  (addr-stable nil)
                                  (data-stable nil)
                                  (dtack- 'DTACK-)
                                  (mem-state 0)
                                  (mem-count 'MEM-COUNT)
                                  (mem-dtack 'MEM-DTACK)
                                  (last-rw- 'T)
                                  (hyps nil)
                                  (enable nil)
                                  (disable nil)
                                  (split-term nil)
                                  (dtack-wait nil))
  (declare (ignore enable))
  (let* ((time-fn (unstring "T_" start-state "->" end-state))
         (time (eval `(,time-fn)))
         (lemma (unstring start-state "->" end-state "$SIM" suffix))
         (term
          `(RUN-FM9001
            (LIST
             (LIST REGS FLAGS
                   A-REG B-REG I-REG
                   DATA-OUT ADDR-OUT
                   RESET- ,dtack- HOLD-
                   PC-REG
                   (,(cv-name start-state)
                    ,last-rw- LAST-REGS-ADDRESS LAST-I-REG
                    LAST-FLAGS PC-REG))
             (LIST MEM ,mem-state CLOCK ,mem-count ,mem-dtack
                   ,last-rw-
                   ,(if addr-stable 'ADDR-OUT 'LAST-ADDRESS)
                   ,(if data-stable 'DATA-OUT 'LAST-DATA)))
            INPUTS
            ,time))
         (hyps
         `(AND (CV-HYPS ,last-rw- LAST-REGS-ADDRESS LAST-I-REG LAST-FLAGS
                        PC-REG)
               (CV-HYPS T PC-REG I-REG FLAGS PC-REG)
               (MEMORY-OKP 32 32 MEM)
               (REGFILE-OKP REGS)
               (BVP A-REG)
               (EQUAL (LENGTH A-REG) 32)
               (BVP B-REG)
               (EQUAL (LENGTH B-REG) 32)
               (BVP ADDR-OUT)
               (EQUAL (LENGTH ADDR-OUT) 32)
               (BVP DATA-OUT)
               (EQUAL (LENGTH DATA-OUT) 32)
               (boolp ,dtack-)
               (boolp HOLD-)
               (RUN-INPUTS-P INPUTS ,time)
               (EQUAL RESET- T)
```

```
          ,hyps))
      (hints
       `(                            ;(no-built-in-arith t)
         (enable regfile-okp)
         (DISABLE PLUS-ADD1 MAKE-TREE *1*MAKE-TREE V-INC V-DEC BV
                  F-BUF ,@disable))))

  `(#+axioms
    ADD-AXIOM
    #-axioms
    PROVE-LEMMA ,lemma (REWRITE)
    (IMPLIES
     ,(if dtack-wait
          `(AND (EQUAL DTACK-WAIT ,dtack-wait)
                ,hyps)
        hyps)
     (EQUAL ,term
            ,(if dtack-wait
                 (print (expand term
                                `(AND (EQUAL DTACK-WAIT ,dtack-wait)
                                      (NOT (ZEROP DTACK-WAIT))
                                      ,hyps)
                                hints))
               (if split-term
                   `(IF ,split-term
                        ,(print (expand term `(AND ,hyps ,split-term) hints))
                      ,(print (expand term `(AND ,hyps (NOT ,split-term))
                                      hints)))
                 (print (expand term hyps hints))))))
    ;;Hint
    #-axioms
    ,(if dtack-wait
         `((INDUCT (ZEROP-NOT-ZEROP-CASES DTACK-WAIT))
           ,@hints)
       hints)
    )))
```

14.13 "monotonicity-macros.lisp"


```
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    MONOTONICITY-MACROS.LISP
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(in-package "USER")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;    MONOTONICITY LEMMAS FOR BOOLEAN FUNCTIONS     ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; e.g.:

;;;   >(macroexpand-1 '(monotonicity-lemma f-and3))
;;;   (PROVE-LEMMA F-AND3-MONOTONE (REWRITE)
;;;       (IMPLIES (AND (B-APPROX A1 A2) (B-APPROX B1 B2) (B-APPROX C1 C2))
;;;                (B-APPROX (F-AND3 A1 B1 C1) (F-AND3 A2 B2 C2))))
;;;   T
;;;
;;;   >


(defun monotonicity-lemma-fn (name &optional hints)
  (let* ((ev (get name 'event))
         (args (caddr ev))
         (args1
          (iterate for arg in args
                   collect (pack (list arg 1))))
         (args2
          (iterate for arg in args
                   collect (pack (list arg 2))))
         (conjuncts
          (iterate for arg1 in args1
                   as arg2 in args2
                   collect (list 'b-approx arg1 arg2))))
    (if args
        `(prove-lemma ,(pack (list name '-monotone)) (rewrite)
                      (implies
                       ,(if (consp (cdr args))
                            (cons 'and conjuncts)
                          (car conjuncts))
                       (b-approx (,name ,@args1) (,name ,@args2)))
                      ,@(and hints (list hints)))
      t)))
```

```
(defmacro monotonicity-lemma (name &optional hints)
  (monotonicity-lemma-fn name hints))


(defmacro monotonicity-lemmas (names &optional hints)
  (list 'do-events-recursive
        (list 'quote
              (iterate for name in names
                       collect
                       (monotonicity-lemma-fn name hints)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                    DISABLE-ALL                  ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defmacro disable-all (&rest names)
  (list 'do-events-recursive
        (list 'quote
              (iterate for name in names
                       collect
                       (list 'disable name)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;           PROVE-PRIMITIVE-MONOTONICITY           ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; >(macroexpand-1 '(prove-primitive-monotonicity (AO2 AO4)))
;;; (DO-EVENTS-RECURSIVE
;;;     '((PROVE-LEMMA DUAL-EVAL-AO2-VALUE (REWRITE)
;;;           (EQUAL (DUAL-EVAL 0 'AO2 ARGS STATE NETLIST)
;;;                  (LET ((A (CAR ARGS)) (B (CAR (CDR ARGS)))
;;;                        (C (CAR (CDR (CDR ARGS))))
;;;                        (D (CAR (CDR (CDR (CDR ARGS))))))
;;;                    (CONS (F-NOR (F-AND A B) (F-AND C D)) 'NIL)))
;;;           ((ENABLE DUAL-EVAL DUAL-APPLY-VALUE)))
;;;       (PROVE-LEMMA DUAL-EVAL-AO2-STATE (REWRITE)
;;;           (EQUAL (DUAL-EVAL 2 'AO2 ARGS STATE NETLIST) 0)
;;;           ((ENABLE DUAL-EVAL DUAL-APPLY-STATE)))
;;;       (PROVE-LEMMA AO2-MONOTONE (REWRITE)
;;;           (AND (MONOTONICITY-PROPERTY 0 'AO2 NETLIST A1 A2 S1 S2)
;;;                (MONOTONICITY-PROPERTY 2 'AO2 NETLIST A1 A2 S1 S2))
;;;           ((DISABLE-THEORY T)
;;;            (ENABLE-THEORY GROUND-ZERO MONOTONICITY-LEMMAS)
;;;            (ENABLE *1*B-APPROX *1*V-APPROX *1*S-APPROX V-APPROX
;;;                    MONOTONICITY-PROPERTY-OPENER-0
;;;                    MONOTONICITY-PROPERTY-OPENER-2 DUAL-EVAL-AO2-VALUE
;;;                    DUAL-EVAL-AO2-STATE S-APPROX-IMPLIES-B-APPROX
;;;                    FOURP-IMPLIES-S-APPROX-IS-B-APPROX FOURP-F-BUF
;;;                    FOURP-F-IF)
;;;            (EXPAND (V-APPROX A1 A2) (V-APPROX (CDR A1) (CDR A2))
;;;                    (V-APPROX (CDR (CDR A1)) (CDR (CDR A2)))
;;;                    (V-APPROX (CDR (CDR (CDR A1))) (CDR (CDR (CDR A2)))))))
;;;       (PROVE-LEMMA DUAL-EVAL-AO4-VALUE (REWRITE)
;;;           (EQUAL (DUAL-EVAL 0 'AO4 ARGS STATE NETLIST)
;;;                  (LET ((A (CAR ARGS)) (B (CAR (CDR ARGS)))
;;;                        (C (CAR (CDR (CDR ARGS))))
;;;                        (D (CAR (CDR (CDR (CDR ARGS))))))
;;;                    (CONS (F-NAND (F-OR A B) (F-OR C D)) 'NIL)))
;;;           ((ENABLE DUAL-EVAL DUAL-APPLY-VALUE)))
;;;       (PROVE-LEMMA DUAL-EVAL-AO4-STATE (REWRITE)
;;;           (EQUAL (DUAL-EVAL 2 'AO4 ARGS STATE NETLIST) 0)
;;;           ((ENABLE DUAL-EVAL DUAL-APPLY-STATE)))
;;;       (PROVE-LEMMA AO4-MONOTONE (REWRITE)
;;;           (AND (MONOTONICITY-PROPERTY 0 'AO4 NETLIST A1 A2 S1 S2)
;;;                (MONOTONICITY-PROPERTY 2 'AO4 NETLIST A1 A2 S1 S2))
;;;           ((DISABLE-THEORY T)
;;;            (ENABLE-THEORY GROUND-ZERO MONOTONICITY-LEMMAS)
;;;            (ENABLE *1*B-APPROX *1*V-APPROX *1*S-APPROX V-APPROX
;;;                    MONOTONICITY-PROPERTY-OPENER-0
```

```
;;;                   MONOTONICITY-PROPERTY-OPENER-2 DUAL-EVAL-A04-VALUE
;;;                   DUAL-EVAL-A04-STATE S-APPROX-IMPLIES-B-APPROX
;;;                   FOURP-IMPLIES-S-APPROX-IS-B-APPROX FOURP-F-BUF
;;;                   FOURP-F-IF)
;;;             (EXPAND (V-APPROX A1 A2) (V-APPROX (CDR A1) (CDR A2))
;;;                   (V-APPROX (CDR (CDR A1)) (CDR (CDR A2)))
;;;                   (V-APPROX (CDR (CDR (CDR A1)))
;;;                            (CDR (CDR (CDR A2))))))))
;;; T
;;;
;;; >


(defun dual-eval-name-state* (name)
  (pack (list 'dual-eval- name '-state)))


(defun dual-eval-name-value* (name)
  (pack (list 'dual-eval- name '-value)))


(defun name-value-let-bindings (inputs)
  (iterate for i in inputs
           with x = 'args
           collect
           (prog1 (list i (list 'car x))
             (setq x (list 'cdr x)))))


(defun dual-eval-state-lemma (name inputs states new-states)
  `(prove-lemma ,(dual-eval-name-state* name) (rewrite)
                (equal (dual-eval 2 ',name args state netlist)
                       ,(if states
                            `(let ,(name-value-let-bindings inputs)
                               ,new-states)
                          0))
                ((enable dual-eval dual-apply-state))))


(defun dual-eval-value-lemma (name inputs states results)
  (declare (ignore states))
  `(prove-lemma ,(dual-eval-name-value* name) (rewrite)
                (equal (dual-eval 0 ',name args state netlist)
                       (let ,(name-value-let-bindings inputs)
                         ,results))
                ((enable dual-eval dual-apply-value))))
```

```
(defun device-monotonicity-lemma-expand-hint (name inputs states)
  (declare (ignore states))
  (cons 'expand
        (iterate for x in inputs
                 with a1 = 'a1
                 and a2 = 'a2
                 collect
                 (prog1
                     (list 'v-approx a1 a2)
                     (ignore-variable x)
                     (setq a1 (list 'cdr a1))
                     (setq a2 (list 'cdr a2))))))


(defun device-monotonicity-lemma (name inputs states)
  `(prove-lemma
    ,(pack (list name '-monotone))
    (rewrite)
    (and (monotonicity-property 0 ',name netlist a1 a2 s1 s2)
         (monotonicity-property 2 ',name netlist a1 a2 s1 s2))
    ((disable-theory t)
     (enable-theory ground-zero monotonicity-lemmas)
     (enable *1*b-approx *1*v-approx *1*s-approx v-approx
             monotonicity-property-opener-0 monotonicity-property-opener-2
             ,(dual-eval-name-value* name) ,(dual-eval-name-state* name)
             s-approx-implies-b-approx
             fourp-implies-s-approx-is-b-approx fourp-f-buf fourp-f-if)
     ,(device-monotonicity-lemma-expand-hint name inputs states))))


(defun prove-primitive-monotonicity-events (name)
  (let ((entry (cdr (assoc name common-lisp-primp-database))))
    (let ((inputs (cdr (assoc 'inputs entry)))
          ;; (outputs (cdr (assoc 'outputs entry)))
          (results (cdr (assoc 'results entry)))
          ;; the following appears to always be 'state or nil
          (states (cdr (assoc 'states entry)))
          (new-states (cdr (assoc 'new-states entry))))
      (list (dual-eval-value-lemma name inputs states results)
            (dual-eval-state-lemma name inputs states new-states)
            (device-monotonicity-lemma name inputs states)))))


(defmacro prove-primitive-monotonicity (x)
  (let ((lst (if (consp x) x (list x))))
    (list 'do-events-recursive
          (list 'quote
                (iterate for name in lst
                         nconc (prove-primitive-monotonicity-events name))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;       REVERTING THE DISABLED/ENABLED STATE      ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defun old-state-events (ev-name)
  (let (on-off)
    (iterate for x in chronology
      until (eq x ev-name)
      with ans and z and ev
      do
      (setq ev (get x 'event))
      (when (and (eq (car ev) 'prove-lemma)
                 (member-eq 'rewrite (caddr ev)))
        (setq ans (cons `(disable ,x) ans)))
      (when (match ev (toggle & z on-off))
        (if on-off
            (setq ans (cons `(enable ,z) ans))
          (setq ans (cons `(disable ,z) ans))))
      finally (return ans))))
```

```
(defmacro revert-state (ev-name)
  `(do-events-recursive ',(old-state-events ev-name)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;    PROVE-DUAL-APPLY-VALUE-DP-RAM-16X32-LEMMA-2   ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; This little hack is simply to save room in the files.

;;; (PROVE-LEMMA DUAL-APPLY-VALUE-DP-RAM-16X32-LEMMA-2 (REWRITE)
;;;     (EQUAL (DUAL-APPLY-VALUE 'DP-RAM-16X32 A S)
;;;            (DUAL-PORT-RAM-VALUE 32 4
;;;                (LIST (EVAL$ T 'READ-A0
;;;                             (APPEND (PAIRLIST
;;;                                      '(READ-A0 READ-A1 READ-A2 READ-A3
;;;                                        WRITE-B0 WRITE-B1 WRITE-B2
;;;                                        WRITE-B3 WEN D0 D1 D2 D3 D4 D5
;;;                                        D6 D7 D8 D9 D10 D11 D12 D13 D14
;;;                                        D15 D16 D17 D18 D19 D20 D21 D22
;;;                                        D23 D24 D25 D26 D27 D28 D29 D30
;;;                                        D31)
;;;                                      A)
;;;                                     (PAIRSTATES 'STATE S)))
;;;                      (EVAL$ T 'READ-A1
;;;                             (APPEND (PAIRLIST
;;;                                      '(READ-A0 READ-A1 READ-A2 READ-A3
;;;                                        WRITE-B0 WRITE-B1 WRITE-B2
;;;                                        WRITE-B3 WEN D0 D1 D2 D3 D4 D5
;;;                                        D6 D7 D8 D9 D10 D11 D12 D13 D14
;;;                                        D15 D16 D17 D18 D19 D20 D21 D22
;;;                                        D23 D24 D25 D26 D27 D28 D29 D30
;;;                                        D31)
;;;                                      A)
;;;                                     (PAIRSTATES 'STATE S)))
;;;                      ....
;;;                      (EVAL$ T 'D31
;;;                             (APPEND (PAIRLIST
;;;                                      '(READ-A0 READ-A1 READ-A2 READ-A3
;;;                                        WRITE-B0 WRITE-B1 WRITE-B2
;;;                                        WRITE-B3 WEN D0 D1 D2 D3 D4 D5
;;;                                        D6 D7 D8 D9 D10 D11 D12 D13 D14
;;;                                        D15 D16 D17 D18 D19 D20 D21 D22
;;;                                        D23 D24 D25 D26 D27 D28 D29 D30
;;;                                        D31)
;;;                                      A)
;;;                                     (PAIRSTATES 'STATE S))))
;;;                (EVAL$ T 'STATE
;;;                       (APPEND (PAIRLIST
;;;                                '(READ-A0 READ-A1 READ-A2 READ-A3
;;;                                  WRITE-B0 WRITE-B1 WRITE-B2 WRITE-B3
;;;                                  WEN D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
;;;                                  D10 D11 D12 D13 D14 D15 D16 D17 D18
;;;                                  D19 D20 D21 D22 D23 D24 D25 D26 D27
;;;                                  D28 D29 D30 D31)
```

```
;;;                                  A)
;;;                              (PAIRSTATES 'STATE S)))))
;;;     ((DISABLE-THEORY T)
;;;      (ENABLE *1*PRIMP *1*LOOKUP-MODULE *1*CDR *1*CAR *1*EVAL$
;;;              DUAL-APPLY-VALUE-DP-RAM-16X32-LEMMA-1 EVAL$-APPEND
;;;              EVAL$-APPEND-2 EVAL$-QUOTE)))


(defmacro prove-dual-apply-value-or-state-dp-ram-16x32-lemma-2 (value-or-state)
  (let ((name1 (if (eq value-or-state 'value)
                   'dual-apply-value-dp-ram-16x32-lemma-2
                 'dual-apply-state-dp-ram-16x32-lemma-2))
        (name2 (if (eq value-or-state 'value)
                   'dual-apply-value
                 'dual-apply-state))
        (name3 (if (eq value-or-state 'value)
                   'dual-port-ram-value
                 'dual-port-ram-state))
        (name4 (if (eq value-or-state 'value)
                   'dual-apply-value-dp-ram-16x32-lemma-1
                 'dual-apply-state-dp-ram-16x32-lemma-1)))
    `(prove-lemma ,name1 (rewrite)
       (equal (,name2 'dp-ram-16x32 a s)
              (,name3
               32 4
               ,(cons 'list
                      (iterate for input in dp-ram-16x32-inputs
                        collect (list 'eval$ 't (list 'quote input)
                                      (list 'append
                                            (list 'pairlist
                                                  (list 'quote
                                                        dp-ram-16x32-inputs)
                                                  'a)
                                            '(pairstates 'state s)))))
               (eval$ t 'state
                      ,(list 'append
                             (list 'pairlist
                                   (list 'quote dp-ram-16x32-inputs)
                                   'a)
                             '(pairstates 'state s)))))
       ((disable-theory t)
        (enable *1*primp *1*lookup-module *1*cdr *1*car *1*eval$
               ,name4 eval$-append rewrite-eval$
               eval$-append-2 eval$-quote)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;    GOOD-STATE LEMMAS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defun device-good-s-lemma (name inputs states)
  (declare (ignore inputs states))
  `(prove-lemma
     ,(pack (list name '-preserves-good-s))
     (rewrite)
     (implies (good-s s)
              (good-s (dual-apply-state ',name args s)))
     ((disable-theory t)
      (enable-theory ground-zero)
      (enable *1*b-approx *1*v-approx *1*s-approx v-approx dual-apply-state
              *1*primp2 f-buf-preserves-good-s f-if-preserves-good-s good-s-0
              ,(dual-eval-name-value* name) ,(dual-eval-name-state* name))
      ;,(device-monotonicity-lemma-expand-hint name inputs states)
      )))


(defun prove-primitive-preserves-good-s-events (name)
  (let ((entry (cdr (assoc name common-lisp-primp-database))))
    (let ((inputs (cdr (assoc 'inputs entry)))
          ;; (outputs (cdr (assoc 'outputs entry)))
          ;; (results (cdr (assoc 'results entry)))
          ;; the following appears to always be 'state or nil
          ;; (new-states (cdr (assoc 'new-states entry)))
          (states (cdr (assoc 'states entry))))
      (list ;(dual-eval-value-lemma name inputs states results)
            ;(dual-eval-state-lemma name inputs states new-states)
            (device-good-s-lemma name inputs states)))))


(defmacro prove-primitive-preserves-good-s (x)
  (let ((lst (if (consp x) x (list x))))
    (list 'do-events-recursive
          (list 'quote
                (iterate for name in lst
                  nconc
                  (prove-primitive-preserves-good-s-events name))))))
```

14.14 "translate.lisp"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    TRANSLATE.LISP
;;;
;;;    A DUAL-EVAL HDL to LSI Logic NDL translator.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(in-package "USER")

;;;
;;; The netlist arg to functions PRINT-NDL-FORM-TO-FILE and PRINT-NDL-FORM
;;; should look like (LISP-NETLIST <netlist>), where <netlist> is a well-formed
;;; netlist.  Recall that LISP-NETLIST changes (INDEX name num) names to
;;; 'name_num names. Make the netlist arg in r-loop by doing
;;;
;;;     (SETQ var (LISP-NETLIST <netlist>))
;;;
;;; Outside of r-loop, the netlist arg will be
;;;
;;;     (CDR (ASSOC 'var R-ALIST))
;;;
;;;
;;; For example, to translate a the entire FM9001 netlist stored in Nqthm as
;;; (CHIP$NETLIST), do the following:
;;;
;;; > (r-loop)
;;; *(SETQ NETLIST (LISP-NETLIST (CHIP$NETLIST)))
;;; ... <r-loop prints the netlist> ..
;;; *OK
;;; > (PRINT-NDL-FORM-TO-FILE (CDR (ASSOC 'NETLIST R-ALIST)) "chip.ndl")
;;;
;;; The translated netlist will be written to "chip.ndl"
;;;
;;; This process defines certain modules that are assumed to be primitives of
;;; our netlist language.


;;;  PRINT-NDL-FORM-TO-FILE netlist file-name
;;;
;;;  This is the top-level call to the translator.  The NDL form of the NETLIST
;;;  will be written to file FILE-NAME.


(defun print-NDL-form-to-file (netlist file-name)
  (with-open-file (stream file-name :direction :output)
    (print-NDL-form netlist stream)))
```

```
(defconstant NDL-linelength 78)


(defun args-format (indent-column
                    &optional (extra-chars 1.)
                              (line-length NDL-linelength))
  (format nil "~~{~~<~~%~~~AT~~~A,~A:;~~A~~>~~^,~~}"
          indent-column extra-chars line-length))

;;;  PRINT-NDL-FORM (netlist &optional (stream *standard-output*))
;;;
;;;  Formats the netlist to a stream.
```

```
(defun print-NDL-form (netlist &optional (stream *standard-output*))
  (let ((m-ins-format (args-format 7))
        (m-outs-format (args-format 8)))
    (format stream "COMPILE;~%DIRECTORY MASTER;~%")
    (dolist (module netlist)
      (let ((m-name (car module))
            (m-ins  (cadr module))
            (m-outs (caddr module))
            (m-occs (cadddr module)))
        (format stream "~%MODULE ~A;~%INPUTS ~@?;~%OUTPUTS ~@?;~%~
                        LEVEL FUNCTION;~%DEFINE~%"
                m-name
                m-ins-format (NDL-name-list m-ins)
                m-outs-format (NDL-name-list m-outs))
        (dolist (occ m-occs)
          (multiple-value-bind (o-name o-outs o-fn o-ins)
            (NDL-occurrence occ)
            (let ((output (format nil
                                  "~A(~{~A~^,~}) = ~A(~{~A~^,~});"
                                  o-name o-outs o-fn o-ins)))
              (if (<= (length output) NDL-linelength)
                  (format stream "~A~%" output)
                (let ((o-name-length (length (princ-to-string o-name)))
                      (o-fn-length (length (princ-to-string o-fn))))
                  (format stream "~A(~@?)~%  = ~A(~@?);~%"
                          o-name
                          (args-format (1+ o-name-length)) o-outs
                          o-fn
                          (args-format (+ 5 o-fn-length) 2) o-ins))))))
        (format stream "END MODULE;~%")))
    (format stream "
MODULE RAM-ENABLE-CIRCUIT;
INPUTS CLK,TEST-REGFILE-,DISABLE-REGFILE-,WE;
OUTPUTS Z;
LEVEL FUNCTION;
DEFINE
G0(CLK-10) = DEL10(CLK);
G1(TEST-REGFILE) = IVA(TEST-REGFILE-);
G2(GATE-CLK) = OR2(CLK-10,TEST-REGFILE);
G3(Z) = ND3P(WE,DISABLE-REGFILE-,GATE-CLK);
END MODULE;

MODULE ID;
INPUTS A;
OUTPUTS Z;
LEVEL FUNCTION;
DEFINE
Z = (A);
END MODULE;

MODULE VDD;
OUTPUTS Z;
LEVEL FUNCTION;
DEFINE
```

```
GO(Z) = ID(NC/1/);
END MODULE;

MODULE VSS;
OUTPUTS Z;
LEVEL FUNCTION;
DEFINE
GO(Z) = ID(NC/0/);
END MODULE;

END COMPILE;
END;
")))

;;;  NDL-NAME x
;;;
;;;  NDL-NAME converts a name from a literal atom to a string, and substitutes
;;;  periods "." for underscores "_".


(defun NDL-name (x)
  (if (symbolp x)
      (substitute #\. #\_ (string x) :count 1 :from-end t)
      (progn (cerror "Return the argument."
                     "The argument ~A is not a symbol." x)
             x)))


(defun NDL-name-list (x)
  (if (consp x)
      (cons (NDL-name (car x))
            (NDL-name-list (cdr x)))
      x))

;;;  NDL-OCCURRENCE occ
;;;
;;;  Translates HDL occurrences to NDL, substituting NDL macrocell names for
;;;  HDL primitive names.
```

```
(defun NDL-occurrence (occ)
  (let* ((o-name (NDL-name (car occ)))
         (o-outs (NDL-name-list (cadr occ)))
         (o-fn (caddr  occ))
         (o-ins (NDL-name-list (cadddr occ)))
         (primp (assoc o-fn common-lisp-primp-database)))
    (if (consp primp)
        ;;  This is a primitive.  What these successive LET* forms are doing is
        ;;  checking for the case where the primitive macrocell has a
        ;;  different ordering on the inputs, in which case the generated
        ;;  macrocell call has to reorder the HDL inputs.  Also, LSI Logic pad
        ;;  macrocells are named "&<something>".  These are not legal Nqthm
        ;;  litatoms.  So, we add the "&" here.
        (let* ((lsi-entry (assoc 'lsi-name (cdr primp)))
               (lsi-value (if (consp lsi-entry) (cdr lsi-entry) o-fn))
               (o-fn2 (if (consp lsi-value) (car lsi-value) lsi-value))
               (o-ins2 (if (consp lsi-value)
                           (sublis (pairlis (cdr (assoc 'inputs (cdr primp)))
                                            o-ins)
                                   (cdr lsi-value))
                           o-ins))
               (o-fn3 (if (assoc 'pads (cdr primp))
                          (concatenate 'string "&" (string o-fn2))
                          o-fn2)))
          (values o-name o-outs o-fn3 o-ins2))
        (values o-name o-outs o-fn o-ins))))
```

14.15 "purify.lisp"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;  Work of Matt Kaufmann.


(IN-PACKAGE "USER")


(EVAL-WHEN (LOAD EVAL COMPILE)
           (CHK-BASE-AND-PACKAGE-1992 10 *PACKAGE*))
```

```
(defparameter *mods-alist*
  (if (boundp '*mods-alist*)            ;so it can work with Nqthm-1987
      (if (not (assoc-eq 'purify *mods-alist*))
          (cons (cons 'purify
                      "A facility for creating event files from
libraries, even when those libraries were created using
various hacks rather than a `pure' Nqthm or Pc-Nqthm.
See \"purify.doc\" for documentation.")
                *mods-alist*)
        *mods-alist*)
    nil))

;; The purpose of this utility is to allow one to take a completed
;; proof effort using (pc-)nqthm and "purify" it a sense I'll now
;; describe.

;; Nqthm users commonly write macros that generate events.  Also,
;; nqthm users at CLI tend to use the "saved_pc-nqthm" core image,
;; which includes local modifications to pc-nqthm.  In some cases
;; there's really no use of pc-nqthm at all beyond what's in nqthm,
;; except perhaps for theories; hence I've also created a file
;; "deftheory.lisp" that contains the disabledp patch and deftheory,
;; but only these.  (Probably I could have left out the disabledp
;; patch, but I didn't want to think about the ramifications that
;; would have for deftheory.)

;; At any rate, once one chooses a core image that corresponds to
;; one of the the publicly released systems -- i.e. either nqthm
;; or pure-pc-nqthm -- one might like to specify some additional
;; forms to execute (perhaps to load files like "deftheory" or
;; "fast-clausifier") and also a library to replay.  The effect should
;; be as illustrated below.

;; 1. The library is used to create an events file.

;; 2. Nqthm or pure-pc-nqthm is started up, with the indicated forms
;;    executed.

;; 3. A proveall is run using the events file created in the first step.

;; A sample transcript occurs at the end of the file.

;; Modifications by B. Brock, July 1993.
;;
;; This now simply creates a file of events, NOT a PROVEALL.
;;
;; The FM9001 proof contains several occurrences of COMPILE-UNCOMPILED-DEFNS,
;; of which there is no trace in the final chronology.  I defined a new macro,
;; COMPILE-UNCOMPILED-DEFNS*, and modified Matt's LIBRARY-TO-EVENTS, to handle
;; this problem.  This macro introduces a new Nqthm function called
;; COMPILE-UNCOMPILED-DEFNS, whose ENABLE events are recognized and used as a
;; tag to insert calls to COMPILE-UNCOMPILED-DEFNS.
;;
;; We also automatically add the (SETQ REDUCE-TERM-CLOCK 2000) form.
```

```
(defmacro library-to-events (filename
                             &optional event-filename events-list-name
                             &rest forms)
  ;; e.g. (library-to-events "foo" "foo.events") should write to
  ;; "foo.events" a list of events that could be run in order to create
  ;; the library "foo", i.e. "foo.lib" and "foo.lisp".  This should work
  ;; even if the filenames have directory prefixes.
  (setq event-filename
        (or event-filename
            (concatenate 'string filename "-replay.events")))
  (setq events-list-name
        (or events-list-name
            (concatenate 'string filename "-replay")))
  `(progn
     (note-lib ,filename)
     (database-to-events ,event-filename ,events-list-name ',forms)))
```

```
(defun database-to-events (event-filename events-list-name forms)
  (declare (ignore events-list-name))
  ;; Writes out a list of events to event-filename that could be run
  ;; in order to create the current prover state.
  (or (equal ".events"
             (subseq event-filename
                     (- (length event-filename) 7)
                     (length event-filename)))
      (error "The first argument to database-to-events should be a ~
              string that ends in \".events\", ~%but ~a does not."
             event-filename))
  (with-open-file
    (str event-filename :direction :output)
    (format t "~%Creating events file ~s...." event-filename)
    (when forms
      (format str "~s~%~%" (if (cdr forms)
                               (cons 'progn forms)
                             (car forms)) str))
    (ppr (quote (boot-strap nqthm)) str)
    (iterpri str)
    (iterpri str)
    (ppr (quote (setq reduce-term-clock 2000)) str)
    (iterpri str)
    (iterpri str)
    ;;  CAR is (BOOT-STRAP NQTHM) -- added above before (SETQ ... )
    (iterate for name in (cdr (reverse chronology))
      do
      (let (event)
        (progn
          (setf event (untranslate-event (get name (quote event))))
          (ppr event str)
          (iterpri str)
          (iterpri str)
          (when (and
                 (equal (car event) (quote toggle))
                 (equal (caddr event) (quote compile-uncompiled-defns))
                 (equal (cadddr event) nil))
            (ppr (quote (compile-uncompiled-defns "tmp")) str)
            (iterpri str)
            (iterpri str)))))
    (print `(make-lib ,(subseq event-filename 0
                               (- (length event-filename) 7)) t) str)
    (format t " done.~%")))
```

```
(defmacro compile-uncompiled-defns* (filename)
  `(PROGN
    ,(let ((sdefn (get 'compile-uncompiled-defns 'sdefn)))
       (if sdefn
           (unless (equal sdefn '(LAMBDA () '*1*TRUE))
             (error "~%COMPILE-UNCOMPILED-DEFNS is already defined!"))
         '(DEFN COMPILE-UNCOMPILED-DEFNS () T)))
    (ENABLE COMPILE-UNCOMPILED-DEFNS)
    (COMPILE-UNCOMPILED-DEFNS ,filename)))

#|

Sample transcript.  It assumes that we used pc-nqthm on the following
events to create library "foo", i.e. "foo.lib" and "foo.lisp" in that
same directory.


(do-events '(


(prove-lemma times-comm (rewrite)
  (equal (times x z) (times z x)))


(defn blob (x)
  (let ((y (plus x x)))
    (times y 3)))


(prove-lemma times-add1 (rewrite)
  (equal (times (add1 x) y)
         (plus y (times x y))))
```

```
(prove-lemma blob-val nil
  (equal (blob x)
         (cond ((zerop x) 0)
               (t (times 6 x)))))
))

.... and, here's what we do to create a pure nqthm events file
"foo-replay.events":

% pc-nqthm
AKCL (Austin Kyoto Common Lisp)  Version(1.527) Mon Jan  7 12:51:51 CST 1991
Contains Enhancements by W. Schelter

Nqthm, with functional instantiation.
Initialized with (BOOT-STRAP NQTHM) on January 7, 1991  14:22:37.
>(load "purify.o")
Loading purify.o
start address -T 30b800 Finished loading purify.o
1864

>(library-to-events "foo" nil nil (print "hi") (print "there"))
To print patch messages, (PRINT-NQTHM-PATCH-DISCLAIMER)
Loading foo.lib
Finished loading foo.lib
Loading foo.lisp
Finished loading foo.lisp

Creating events file "foo-replay.events".... done.
NIL

>(bye)
Bye.
% nqthm
AKCL (Austin Kyoto Common Lisp)  Version(1.527) Mon Jan  7 12:51:51 CST 1991
Contains Enhancements by W. Schelter

Nqthm, with functional instantiation.
Initialized with (BOOT-STRAP NQTHM) on January 7, 1991  14:22:37.
>(load "foo-replay.events")
Loading foo-replay.events

"hi"
"there"
NQTHM,TIMES-COMM,BLOB,TIMES-ADD1,BLOB-VAL,
Total Statistics:
[ 6.6 18.0 1.8 ]
```

```
(MAKE-LIB foo-replay)
Finished loading foo-replay.events
T

>

|#
```

14.16 "bags.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.


(BOOT-STRAP NQTHM)


(DEFN DELETE (X L)
      (IF (LISTP L)
          (IF (EQUAL X (CAR L))
              (CDR L)
              (CONS (CAR L) (DELETE X (CDR L))))
          L))


(DEFN BAGDIFF (X Y)
      (IF (LISTP Y)
          (IF (MEMBER (CAR Y) X)
              (BAGDIFF (DELETE (CAR Y) X) (CDR Y))
              (BAGDIFF X (CDR Y)))
          X))


(DEFN BAGINT (X Y)
      (IF (LISTP X)
          (IF (MEMBER (CAR X) Y)
              (CONS (CAR X)
                    (BAGINT (CDR X) (DELETE (CAR X) Y)))
              (BAGINT (CDR X) Y))
          NIL))


(DEFN OCCURRENCES
      (X L)
      (IF (LISTP L)
          (IF (EQUAL X (CAR L))
              (ADD1 (OCCURRENCES X (CDR L)))
              (OCCURRENCES X (CDR L)))
          0))


(DEFN SUBBAGP (X Y)
      (IF (LISTP X)
          (IF (MEMBER (CAR X) Y)
              (SUBBAGP (CDR X) (DELETE (CAR X) Y))
              F)
          T))
```

```
(LEMMA LISTP-DELETE (REWRITE)
       (EQUAL (LISTP (DELETE X L))
              (IF (LISTP L)
                  (OR (NOT (EQUAL X (CAR L)))
                      (LISTP (CDR L)))
                  F))
       ((ENABLE DELETE)
        (INDUCT (DELETE X L))))


(disable listp-delete)


(LEMMA DELETE-NON-MEMBER (REWRITE)
       (IMPLIES (NOT (MEMBER X Y))
                (EQUAL (DELETE X Y) Y))
       ((ENABLE DELETE)))


(LEMMA DELETE-DELETE (REWRITE)
       (EQUAL (DELETE Y (DELETE X Z))
              (DELETE X (DELETE Y Z)))
       ((ENABLE DELETE DELETE-NON-MEMBER)))


(lemma equal-occurrences-zero (rewrite)
       (equal (equal (occurrences x l) 0)
              (not (member x l)))
       ((enable occurrences)))


(LEMMA MEMBER-NON-LIST (REWRITE)
       (IMPLIES (NOT (LISTP L))
                (NOT (MEMBER X L))))


(lemma member-delete (rewrite)
       (equal (member x (delete y l))
              (if (member x l)
                  (if (equal x y)
                      (lessp 1 (occurrences x l))
                      t)
                  f))
       ((enable delete occurrences)))



(LEMMA MEMBER-DELETE-IMPLIES-MEMBERSHIP (REWRITE)
       (IMPLIES (MEMBER X (DELETE Y L))
                (MEMBER X L))
       ((ENABLE DELETE)))
```

```
(LEMMA OCCURRENCES-DELETE (REWRITE)
       (EQUAL (OCCURRENCES X (DELETE Y L))
              (IF (EQUAL X Y)
                  (IF (MEMBER X L)
                      (SUB1 (OCCURRENCES X L))
                      0)
                  (OCCURRENCES X L)))
       ((ENABLE OCCURRENCES DELETE EQUAL-OCCURRENCES-ZERO)))


(LEMMA MEMBER-BAGDIFF (REWRITE)
       (EQUAL (MEMBER X (BAGDIFF A B))
              (LESSP (OCCURRENCES X B)
                     (OCCURRENCES X A)))
       ((ENABLE BAGDIFF OCCURRENCES EQUAL-OCCURRENCES-ZERO
                OCCURRENCES-DELETE)))


(lemma bagdiff-delete (rewrite)
       (equal (bagdiff (delete e x) y)
              (delete e (bagdiff x y)))
       ((enable BAGDIFF DELETE
                DELETE-DELETE
                DELETE-NON-MEMBER
                MEMBER-BAGDIFF
                MEMBER-DELETE
                OCCURRENCES-DELETE)))



(LEMMA SUBBAGP-DELETE (REWRITE)
       (IMPLIES (SUBBAGP X (DELETE U Y))
                (SUBBAGP X Y))
       ((ENABLE DELETE SUBBAGP DELETE-DELETE
                MEMBER-DELETE-IMPLIES-MEMBERSHIP)))


(LEMMA SUBBAGP-CDR1 (REWRITE)
       (IMPLIES (SUBBAGP X Y)
                (SUBBAGP (CDR X) Y))
       ((ENABLE SUBBAGP SUBBAGP-DELETE)))


(LEMMA SUBBAGP-CDR2 (REWRITE)
       (IMPLIES (SUBBAGP X (CDR Y))
                (SUBBAGP X Y))
       ((ENABLE DELETE SUBBAGP DELETE-NON-MEMBER SUBBAGP-CDR1)))


(LEMMA SUBBAGP-BAGINT1 (REWRITE)
       (SUBBAGP (BAGINT X Y) X)
       ((ENABLE DELETE SUBBAGP BAGINT SUBBAGP-CDR2)))
```

```
(LEMMA SUBBAGP-BAGINT2 (REWRITE)
       (SUBBAGP (BAGINT X Y) Y)
       ((ENABLE SUBBAGP BAGINT SUBBAGP-CDR2)))


(prove-lemma occurrences-bagint
             (rewrite)
             (equal (occurrences x (bagint a b))
                    (if (lessp (occurrences x a)
                               (occurrences x b))
                        (occurrences x a)
                        (occurrences x b)))
             ((enable occurrences bagint equal-occurrences-zero
                      occurrences-delete)))


(prove-lemma occurrences-bagdiff
             (rewrite)
             (equal (occurrences x (bagdiff a b))
                    (difference (occurrences x a)
                                (occurrences x b)))
             ((enable occurrences bagdiff equal-occurrences-zero
                      occurrences-delete)))


(prove-lemma member-bagint
             (rewrite)
             (equal (member x (bagint a b))
                    (and (member x a) (member x b)))
             ((enable bagint member-delete)))


(deftheory bags
  (occurrences-bagint
   bagdiff-delete
   occurrences-bagdiff
   member-bagint
   member-bagdiff
   subbagp-bagint2
   subbagp-bagint1
   subbagp-cdr2
   subbagp-cdr1
   subbagp-delete))
```

14.17 "naturals.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;; NATURALS Theory

;; Created by Bill Bevier 1988 (see CLI internal note 057)

;; Modifications by Bill Bevier and Matt Wilding (9/89) including
;; adding some new metalemmas for times, reorganizing the theories,
;; removing some extraneous lemmas, and removing dependence upon
;; other theories (by adding the pertinent lemmas).

;; This script requires the bags theory

;; This script sets up a theory for the NATURALS with the following subtheories
;; ADDITION
;; MULTIPLICATION
;; REMAINDER
;; QUOTIENT
;; EXPONENTIATION
;; LOGS
;; GCDS

;; The theories of EXPONENTIATION, LOGS, and GCDS still need a lot of work

; ----------------------------------------------------------------------------
; ARITHMETIC
; ----------------------------------------------------------------------------

; -------------------- PLUS & DIFFERENCE --------------------

; ---------- EQUAL ----------


(lemma equal-plus-0 (rewrite)
       (equal (equal (plus a b) 0)
              (and (zerop a)
                   (zerop b))))


(lemma plus-cancellation (rewrite)
       (equal (equal (plus a b) (plus a c))
              (equal (fix b) (fix c))))


(disable plus-cancellation)
```

```
(lemma equal-difference-0 (rewrite)
       (and (equal (equal (difference x y) 0)
                   (not (lessp y x)))
            (equal (equal 0 (difference x y))
                   (not (lessp y x))))
       ((induct (difference x y))))


(lemma difference-cancellation (rewrite)
       (equal (equal (difference x y) (difference z y))
              (if (lessp x y)
                  (not (lessp y z))
                  (if (lessp z y)
                      (not (lessp y x))
                      (equal (fix x) (fix z)))))
       ((enable equal-difference-0)))


(disable difference-cancellation)

; ---------- PLUS ----------


(lemma commutativity-of-plus (rewrite)
       (equal (plus x y) (plus y x)))


(lemma commutativity2-of-plus (rewrite)
       (equal (plus x (plus y z))
              (plus y (plus x z))))


(lemma plus-zero-arg2 (rewrite)
       (implies (zerop y)
                (equal (plus x y)
                       (fix x)))
       ((induct (plus x y))))


(lemma plus-add1-arg1 (rewrite)
       (equal (plus (add1 a) b)
              (add1 (plus a b))))


(lemma plus-add1-arg2 (rewrite)
       (equal (plus x (add1 y))
              (if (numberp y)
                  (add1 (plus x y))
                  (add1 x))))


(lemma associativity-of-plus  (rewrite)
       (equal (plus (plus x y) z) (plus x (plus y z))))
```

```
(lemma plus-difference-arg1 (rewrite)
       (equal (plus (difference a b) c)
              (if (lessp b a)
                  (difference (plus a c) b)
                  (plus 0 c)))
       ((induct (difference a b))))


(lemma plus-difference-arg2 (rewrite)
       (equal (plus a (difference b c))
              (if (lessp c b)
                  (difference (plus a b) c)
                  (plus a 0)))
       ((induct (plus a b))))

; ---------- DIFFERENCE-PLUS cancellation rules ----------
;
; Here are the basic canonicalization rules for differences of sums. These
; are subsumed by the meta lemmas and are therefore globally disabled.
; They are here merely to prove the meta lemmas.


(lemma difference-plus-cancellation-proof ()
       (equal (difference (plus x y) x) (fix y)))



(lemma difference-plus-cancellation (rewrite)
       (and (equal (difference (plus x y) x) (fix y))
            (equal (difference (plus y x) x) (fix y)))
       ((use (difference-plus-cancellation-proof (x x) (y y)))
        (enable commutativity-of-plus)))


(disable difference-plus-cancellation)


(lemma difference-plus-plus-cancellation-proof ()
       (equal (difference (plus x y) (plus x z))
              (difference y z)))


(lemma difference-plus-plus-cancellation (rewrite)
       (and (equal (difference (plus x y) (plus x z))
                   (difference y z))
            (equal (difference (plus y x) (plus x z))
                   (difference y z))
            (equal (difference (plus x y) (plus z x))
                   (difference y z))
            (equal (difference (plus y x) (plus z x))
                   (difference y z)))
       ((use (difference-plus-plus-cancellation-proof (x x) (y y) (z z)))
        (enable commutativity-of-plus)))
```

```
(disable difference-plus-plus-cancellation)


(lemma difference-plus-plus-cancellation-hack (rewrite)
       (equal (difference (plus w x a) (plus y z a))
              (difference (plus w x) (plus y z)))
       ((enable commutativity-of-plus
                commutativity2-of-plus
                difference-plus-plus-cancellation)
        (do-not-induct t)))


(disable difference-plus-plus-cancellation-hack)

; Here are a few more facts about difference needed to prove the meta lemmas.
; These are disabled here. We re-prove them after the proof of the meta
; lemmas so that they will fire before the meta lemmas in subsequent proofs.


(lemma diff-sub1-arg2 (rewrite)
       (equal (difference a (sub1 b))
              (if (zerop b)
                  (fix a)
                  (if (lessp a b)
                      0
                      (add1 (difference a b)))))
       ((induct (difference a b))))


(disable diff-sub1-arg2)


(lemma diff-diff-arg1 (rewrite)
    (equal (difference (difference x y) z)
           (difference x (plus y z))))



(lemma diff-diff-arg2 (rewrite)
       (equal (difference a (difference b c))
              (if (lessp b c)
                  (fix a)
                  (difference (plus a c) b)))
       ((enable diff-sub1-arg2
                plus-zero-arg2)
        (induct (difference a b))))


; diff-diff-diff should be removed, but since the hack lemmas for
; correctness-of-cancel-difference-plus are designed for it, we'll
; keep it around.
```

```
(lemma diff-diff-diff (rewrite)
       (implies (and (leq b a)
                     (leq d c))
                (equal (difference (difference a b)
                                   (difference c d))
                       (difference (plus a d) (plus b c))))
       ((enable diff-diff-arg1
                diff-diff-arg2
                plus-difference-arg2
                plus-zero-arg2)
        (do-not-induct t)))


(disable diff-diff-diff)


(lemma difference-lessp-arg1 (rewrite)
       (implies (lessp a b)
                (equal (difference a b) 0)))


(disable difference-lessp-arg1)

; ----------------------------------------------------------------------------
; Meta Lemmas to Cancel PLUS and DIFFERENCE expressions
; ----------------------------------------------------------------------------

; ---------- PLUS-TREE and PLUS-FRINGE ----------


(defn plus-fringe (x)
  (if (and (listp x) (equal (car x) 'plus))
      (append (plus-fringe (cadr x)) (plus-fringe (caddr x)))
      (cons x nil)))


(defn plus-tree (l)
  (if (nlistp l)
      ''0
        (if (nlistp (cdr l))
            (list 'fix (car l))
            (if (nlistp (cddr l))
                (list 'plus (car l) (cadr l))
                (list 'plus (car l) (plus-tree (cdr l)))))))


(lemma numberp-eval$-plus (rewrite)
       (implies (and (listp x) (equal (car x) 'plus))
                (numberp (eval$ t x a))))


(disable numberp-eval$-plus)
```

```
(lemma numberp-eval$-plus-tree (rewrite)
       (numberp (eval$ t (plus-tree l) a)) ((enable plus-tree)))


(disable numberp-eval$-plus-tree)


(lemma member-implies-plus-tree-greatereqp (rewrite)
       (implies (member x y)
                (not (lessp (eval$ t (plus-tree y) a) (eval$ t x a))))
       ((enable plus-tree
                plus-zero-arg2)))


(disable member-implies-plus-tree-greatereqp)


(lemma plus-tree-delete (rewrite)
       (equal (eval$ t (plus-tree (delete x y)) a)
              (if (member x y)
                  (difference (eval$ t (plus-tree y) a) (eval$ t x a))
                  (eval$ t (plus-tree y) a)))
       ((enable delete plus-tree
                delete-non-member
                difference-plus-cancellation
                equal-difference-0
                equal-plus-0
                listp-delete
                member-implies-plus-tree-greatereqp
                numberp-eval$-plus-tree
                plus-zero-arg2)))


(disable plus-tree-delete)


(lemma subbagp-implies-plus-tree-greatereqp (rewrite)
       (implies (subbagp x y)
                (not (lessp (eval$ t (plus-tree y) a)
                            (eval$ t (plus-tree x) a))))
       ((enable plus-tree subbagp
                member-implies-plus-tree-greatereqp
                plus-tree-delete
                plus-zero-arg2
                subbagp-cdr2)))


(disable subbagp-implies-plus-tree-greatereqp)
```

```
(lemma plus-tree-bagdiff (rewrite)
       (implies (subbagp x y)
                (equal (eval$ t (plus-tree (bagdiff y x)) a)
                       (difference (eval$ t (plus-tree y) a)
                                   (eval$ t (plus-tree x) a))))
       ((enable bagdiff plus-tree subbagp
                commutativity-of-plus
                diff-diff-arg1
                difference-lessp-arg1
                member-implies-plus-tree-greatereqp
                numberp-eval$-plus-tree
                plus-tree-delete
                plus-zero-arg2
                subbagp-cdr2
                subbagp-implies-plus-tree-greatereqp)))

(disable plus-tree-bagdiff)

(lemma numberp-eval$-bridge (rewrite)
       (implies (equal (eval$ t z a) (eval$ t (plus-tree x) a))
                (numberp (eval$ t z a)))
       ((enable plus-tree
                numberp-eval$-plus-tree)))

(disable numberp-eval$-bridge)

(lemma bridge-to-subbagp-implies-plus-tree-greatereqp (rewrite)
       (implies (and (subbagp y (plus-fringe z))
                     (equal (eval$ t z a)
                            (eval$ t (plus-tree (plus-fringe z)) a)))
                (equal (lessp (eval$ t z a) (eval$ t (plus-tree y) a)) f))
       ((enable subbagp plus-fringe plus-tree
                subbagp-implies-plus-tree-greatereqp)))

(disable bridge-to-subbagp-implies-plus-tree-greatereqp)

(lemma eval$-plus-tree-append (rewrite)
       (equal (eval$ t (plus-tree (append x y)) a)
              (plus (eval$ t (plus-tree x) a) (eval$ t (plus-tree y) a)))
       ((enable plus-zero-arg2 commutativity2-of-plus commutativity-of-plus
                equal-plus-0 plus-cancellation plus-tree
                numberp-eval$-plus-tree numberp-eval$-bridge)))

(disable eval$-plus-tree-append)
```

```
(lemma plus-tree-plus-fringe (rewrite)
       (equal (eval$ t (plus-tree (plus-fringe x)) a) (fix (eval$ t x a)))
       ((enable plus-zero-arg2 commutativity-of-plus plus-fringe plus-tree
                numberp-eval$-plus numberp-eval$-bridge
                eval$-plus-tree-append)
        (induct (plus-fringe x))))


(disable plus-tree-plus-fringe)


(lemma member-implies-numberp (rewrite)
       (implies (and (member c (plus-fringe x)) (numberp (eval$ t c a)))
                (numberp (eval$ t x a)))
       ((enable plus-fringe numberp-eval$-plus)
        (induct (plus-fringe x))))


(disable member-implies-numberp)


(lemma cadr-eval$-list (rewrite)
       (and (equal (car (eval$ 'list x a)) (eval$ t (car x) a))
            (equal (cdr (eval$ 'list x a))
                   (if (listp x) (eval$ 'list (cdr x) a) 0))))


(disable cadr-eval$-list)


(lemma eval$-quote (rewrite)
       (equal (eval$ t (cons 'quote args) a) (car args)))


(disable eval$-quote)


(lemma listp-eval$ (rewrite)
       (equal (listp (eval$ 'list x a)) (listp x)))


(disable listp-eval$)

; ---------- CANCEL PLUS ----------

; CANCEL-EQUAL-PLUS cancels identical terms in a term which is the equality
; of two sums. For example,
;
;   (EQUAL (PLUS A B C) (PLUS B D E)) => (EQUAL (PLUS A C) (PLUS D E))
;
```

```
(defn cancel-equal-plus (x)
  (if (and (listp x) (equal (car x) 'equal))
      (if (and (listp (cadr x)) (equal (caadr x) 'plus)
               (listp (caddr x)) (equal (caaddr x) 'plus))
          (list 'equal
                (plus-tree
                 (bagdiff (plus-fringe (cadr x))
                          (bagint (plus-fringe (cadr x))
                                  (plus-fringe (caddr x)))))
                (plus-tree
                 (bagdiff (plus-fringe (caddr x))
                          (bagint (plus-fringe (cadr x))
                                  (plus-fringe (caddr x))))))
          (if (and (listp (cadr x)) (equal (caadr x) 'plus)
                   (member (caddr x) (plus-fringe (cadr x))))
              (list 'if (list 'numberp (caddr x))
                    (list 'equal
                          (plus-tree
                           (delete (caddr x)
                                   (plus-fringe (cadr x))))
                          ''0)
                    (list 'quote f))
              (if (and (listp (caddr x)) (equal (caaddr x) 'plus)
                       (member (cadr x) (plus-fringe (caddr x))))
                  (list 'if (list 'numberp (cadr x))
                        (list 'equal ''0
                              (plus-tree
                               (delete (cadr x)
                                       (plus-fringe (caddr x)))))
                        (list 'quote f))
                  x)))
      x))
```

```
(lemma correctness-of-cancel-equal-plus ((meta equal))
       (equal (eval$ t x a) (eval$ t (cancel-equal-plus x) a))
       ((enable bridge-to-subbagp-implies-plus-tree-greatereqp
                cancel-equal-plus
                difference-cancellation
                equal-difference-0
                eval$-quote
                member-implies-numberp
                member-implies-plus-tree-greatereqp
                numberp-eval$-plus
                plus-tree-bagdiff
                plus-tree-delete
                plus-tree-plus-fringe
                subbagp-bagint1
                subbagp-bagint2)
        (disable eval$)))

; ---------- CANCEL-DIFFERENCE-PLUS ----------

; CANCEL-DIFFERENCE-PLUS cancels identical terms in a term which is the
; difference of two sums. For example,
;
;  (DIFFERENCE (PLUS A B C) (PLUS B D E)) => (DIFFERENCE (PLUS A C) (PLUS D E))
;
; Using rewrite rules, we canonicalize terms involving  PLUS and DIFFERENCE
; to be the DIFFERENCE of two sums. Then CANCEL-DIFFERENCE-PLUS cancels out
; like terms.


(defn cancel-difference-plus (x)
  (if (and (listp x) (equal (car x) 'difference))
      (if (and (listp (cadr x)) (equal (caadr x) 'plus)
               (listp (caddr x)) (equal (caaddr x) 'plus))
          (list 'difference
                (plus-tree
                 (bagdiff (plus-fringe (cadr x))
                          (bagint (plus-fringe (cadr x))
                                  (plus-fringe (caddr x)))))
                (plus-tree
                 (bagdiff (plus-fringe (caddr x))
                          (bagint (plus-fringe (cadr x))
                                  (plus-fringe (caddr x))))))
          (if (and (listp (cadr x)) (equal (caadr x) 'plus)
                   (member (caddr x) (plus-fringe (cadr x))))
              (plus-tree (delete (caddr x) (plus-fringe (cadr x))))
              (if (and (listp (caddr x)) (equal (caaddr x) 'plus)
                       (member (cadr x) (plus-fringe (caddr x))))
                  ''0
                   x)))
      x))
```

```
(lemma correctness-of-cancel-difference-plus ((meta difference))
       (equal (eval$ t x a) (eval$ t (cancel-difference-plus x) a))
       ((enable cancel-difference-plus
                associativity-of-plus
                bridge-to-subbagp-implies-plus-tree-greatereqp
                commutativity-of-plus
                diff-diff-diff
                difference-lessp-arg1
                difference-plus-plus-cancellation-hack
                equal-difference-0
                eval$-quote
                member-implies-plus-tree-greatereqp
                numberp-eval$-plus
                plus-tree-bagdiff
                plus-tree-delete
                plus-tree-plus-fringe
                subbagp-bagint1
                subbagp-bagint2)
        (disable eval$)))

; ---------- DIFFERENCE ----------

; Here are the rules for difference terms which we want to try before
; the meta lemmas. They help canonicalize terms to differences of sums.


(lemma difference-elim (elim)
       (implies (and (numberp y)
                     (not (lessp y x)))
                (equal (plus x (difference y x)) y)))


(lemma difference-leq-arg1 (rewrite)
       (implies (leq a b)
                (equal (difference a b) 0)))


(lemma difference-add1-arg2 (rewrite)
       (equal (difference a (add1 b))
              (if (lessp b a)
                  (sub1 (difference a b))
                  0))
       ((enable difference-leq-arg1)
        (induct (difference a b))))
```

```
(lemma difference-sub1-arg2 (rewrite)
       (equal (difference a (sub1 b))
              (if (zerop b)
                  (fix a)
                  (if (lessp a b)
                      0
                      (add1 (difference a b)))))
       ((enable diff-sub1-arg2)))


(lemma difference-difference-arg1 (rewrite)
       (equal (difference (difference x y) z)
              (difference x (plus y z)))
       ((enable diff-diff-arg1)))


(lemma difference-difference-arg2 (rewrite)
       (equal (difference a (difference b c))
              (if (lessp b c)
                  (fix a)
                  (difference (plus a c) b)))
       ((enable diff-diff-arg2)))


(lemma difference-x-x (rewrite)
       (equal (difference x x) 0))

; ---------- LESSP ----------


(lemma lessp-difference-cancellation (rewrite)
       (equal (lessp (difference a c)
                     (difference b c))
              (if (leq c a)
                  (lessp a b)
                  (lessp c b)))
       ((enable equal-difference-0)))


(disable lessp-difference-cancellation)

; CANCEL-LESSP-PLUS cancels LESSP terms whose arguments are sums.
; Examples:
; (LESSP (PLUS A B C) (PLUS A C D)) -> (LESSP (FIX B) (FIX D))
; (LESSP A (PLUS A B))  -> (NOT (ZEROP (FIX B)))
; (LESSP (PLUS A B) A)  -> F
```

```
(defn cancel-lessp-plus (x)
  (if (and (listp x) (equal (car x) 'lessp))
      (if (and (listp (cadr x)) (equal (caadr x) 'plus)
               (listp (caddr x)) (equal (caaddr x) 'plus))
          (list 'lessp
                (plus-tree
                 (bagdiff (plus-fringe (cadr x))
                          (bagint (plus-fringe (cadr x))
                                  (plus-fringe (caddr x)))))
                (plus-tree
                 (bagdiff (plus-fringe (caddr x))
                          (bagint (plus-fringe (cadr x))
                                  (plus-fringe (caddr x))))))
          (if (and (listp (cadr x)) (equal (caadr x) 'plus)
                   (member (caddr x) (plus-fringe (cadr x))))
              (list 'quote f)
              (if (and (listp (caddr x)) (equal (caaddr x) 'plus)
                       (member (cadr x) (plus-fringe (caddr x))))
                  (list 'not
                        (list 'zerop (plus-tree
                                      (delete (cadr x)
                                              (plus-fringe (caddr x))))))
                  x)))
      x))
```

```
(lemma correctness-of-cancel-lessp-plus ((meta lessp))
       (equal (eval$ t x a) (eval$ t (cancel-lessp-plus x) a))
       ((enable cancel-lessp-plus
                bridge-to-subbagp-implies-plus-tree-greatereqp
                equal-difference-0
                eval$-quote
                lessp-difference-cancellation
                member-implies-plus-tree-greatereqp
                numberp-eval$-plus
                plus-tree-bagdiff
                plus-tree-delete
                plus-tree-plus-fringe
                subbagp-bagint1
                subbagp-bagint2)
        (disable eval$)))


; Define the available theory of addition.  To get the list of events to
; put in the theory, evaluate the following form in NQTHM at this point
; in the script. This form lists all  lemmas which are globally enabled,
; and which have non-null lemma type.
;
; (remove-if-not (function (lambda (x)
;                           (and (member x (lemmas))
;                                (not (assoc x disabled-lemmas))
;                                (not (null (nth 2 (get x 'event)))))))
;               chronology)
```

```
(deftheory addition
  (EQUAL-PLUS-0
   EQUAL-DIFFERENCE-0
   COMMUTATIVITY-OF-PLUS
   COMMUTATIVITY2-OF-PLUS
   PLUS-ZERO-ARG2
   PLUS-ADD1-ARG2
   PLUS-ADD1-ARG1
   ASSOCIATIVITY-OF-PLUS
   PLUS-DIFFERENCE-ARG1
   PLUS-DIFFERENCE-ARG2
   diff-diff-arg1
   diff-diff-arg2
   CORRECTNESS-OF-CANCEL-EQUAL-PLUS
   CORRECTNESS-OF-CANCEL-DIFFERENCE-PLUS
   DIFFERENCE-ELIM
   DIFFERENCE-LEQ-ARG1
   DIFFERENCE-ADD1-ARG2
   DIFFERENCE-SUB1-ARG2
   DIFFERENCE-DIFFERENCE-ARG1
   DIFFERENCE-DIFFERENCE-ARG2
   DIFFERENCE-X-X
   CORRECTNESS-OF-CANCEL-LESSP-PLUS))




; ---------- TIMES ----------


(lemma equal-times-0 (rewrite)
       (equal (equal (times x y) 0)
              (or (zerop x)
                  (zerop y)))
       ((enable equal-plus-0)
        (induct (times x y))))


(lemma equal-times-1 (rewrite)
       (equal (equal (times a b) 1)
              (and (equal a 1)
                   (equal b 1)))
       ((enable equal-plus-0)
        (induct (times a b))))

;(lemma equal-sub1-times-0 (rewrite)
;       (equal (equal (sub1 (times a b)) 0)
;             (or (zerop a)
;                 (zerop b)
;                 (and (equal a 1) (equal b 1)))))
```

```
(lemma equal-sub1-0 (rewrite)
       (equal (equal (sub1 x) 0)
              (or (zerop x)
                  (equal x 1))))


(lemma times-zero (rewrite)
       (implies (zerop y)
                (equal (times x y) 0))
       ((enable plus-zero-arg2 commutativity-of-plus)))


(lemma times-add1 (rewrite)
       (equal (times x (add1 y))
              (if (numberp y)
                  (plus x (times x y))
                  (fix x)))
       ((enable plus-zero-arg2 commutativity-of-plus)))


(lemma commutativity-of-times (rewrite)
       (equal (times y x) (times x y))
       ((enable times-zero times-add1)))


(lemma times-distributes-over-plus-proof ()
       (equal (times x (plus y z))
              (plus (times x y) (times x z)))
       ((enable commutativity2-of-plus
                associativity-of-plus)))


(lemma times-distributes-over-plus (rewrite)
       (and (equal (times x (plus y z))
                   (plus (times x y) (times x z)))
            (equal (times (plus x y) z)
                   (plus (times x z) (times y z))))
       ((use (times-distributes-over-plus-proof (x x) (y y) (z z))
             (times-distributes-over-plus-proof (x z) (y x) (z y)))
        (enable commutativity-of-times)))


(lemma commutativity2-of-times (rewrite)
       (equal (times x y z)
              (times y x z))
       ((enable commutativity-of-times
                times-distributes-over-plus)))
```

```
(lemma associativity-of-times (rewrite)
       (equal (times (times x y) z)
              (times x y z))
       ((enable commutativity-of-times
                commutativity2-of-times)))


(lemma times-distributes-over-difference-proof ()
       (equal (times (difference a b) c)
              (difference (times a c) (times b c)))
       ((enable commutativity-of-times)
        (enable-theory addition)))


(lemma times-distributes-over-difference (rewrite)
       (and (equal (times (difference a b) c)
                   (difference (times a c) (times b c)))
            (equal (times a (difference b c))
                   (difference (times a b) (times a c))))
       ((use (times-distributes-over-difference-proof (a a) (b b) (c c))
             (times-distributes-over-difference-proof (a b) (b c) (c a)))
        (enable commutativity-of-times)))


(lemma times-quotient-proof ()
       (implies (and (not (zerop x))
                     (equal (remainder y x) 0))
                (equal (times (quotient y x) x)
                       (fix y)))
       ((enable times-zero times-add1)
        (induct (remainder y x))))


(lemma times-quotient (rewrite)
       (implies (and (not (zerop y))
                     (equal (remainder x y) 0))
                (and (equal (times (quotient x y) y)
                            (fix x))
                     (equal (times y (quotient x y))
                            (fix x))))
       ((use (times-quotient-proof (x y) (y x)))
        (enable commutativity-of-times)))


(lemma times-1-arg1 (rewrite)
       (equal (times 1 x) (fix x))
       ((enable times-zero)))
```

```
(lemma lessp-times1-proof ()
       (implies (and (lessp a b)
                     (not (zerop c)))
                (equal (lessp a (times b c))
                       t)))


(lemma lessp-times1 (rewrite)
       (implies (and (lessp a b)
                     (not (zerop c)))
                (and (equal (lessp a (times b c))
                            t)
                     (equal (lessp a (times c b))
                            t)))
       ((enable commutativity-of-times)
        (use (lessp-times1-proof (a a) (b b) (c c)))
        (do-not-induct t)))


(lemma lessp-times2-proof ()
       (implies (and (leq a b)
                     (not (zerop c)))
                (equal (lessp (times b c) a)
                       f)))


(lemma lessp-times2 (rewrite)
       (implies (and (leq a b)
                     (not (zerop c)))
                (and (equal (lessp (times b c) a)
                            f)
                     (equal (lessp (times c b) a)
                            f)))
       ((enable commutativity-of-times)
        (use (lessp-times2-proof (a a) (b b) (c c)))
        (do-not-induct t)))


(lemma lessp-times3-proof1 ()
       (implies (and (not (zerop a))
                     (lessp 1 b))
                (lessp a (times a b)))
       ((enable-theory addition)
        (enable times-zero)))


(lemma lessp-times3-proof2 ()
       (implies (lessp a (times a b))
                (and (not (zerop a))
                     (lessp 1 b)))
       ((enable-theory addition)))
```

```
(lemma lessp-times3 (rewrite)
       (and (equal (lessp a (times a b))
                   (and (not (zerop a))
                        (lessp 1 b)))
            (equal (lessp a (times b a))
                   (and (not (zerop a))
                        (lessp 1 b))))
       ((enable commutativity-of-times)
        (use (lessp-times3-proof1 (a a) (b b))
             (lessp-times3-proof2 (a a) (b b)))
        (do-not-induct t)))
```

```
(lemma lessp-times-cancellation-proof ()
       (equal (lessp (times x z) (times y z))
              (and (not (zerop z))
                   (lessp x y)))
       ((enable commutativity-of-times
                correctness-of-cancel-lessp-plus
                times-zero)))
```

```
(lemma lessp-times-cancellation1 (rewrite)
       (and (equal (lessp (times x z) (times y z))
                   (and (not (zerop z))
                        (lessp x y)))
            (equal (lessp (times z x) (times y z))
                   (and (not (zerop z))
                        (lessp x y)))
            (equal (lessp (times x z) (times z y))
                   (and (not (zerop z))
                        (lessp x y)))
            (equal (lessp (times z x) (times z y))
                   (and (not (zerop z))
                        (lessp x y))))
       ((use (lessp-times-cancellation-proof (x x) (y y) (z z)))
        (enable commutativity-of-times)
        (do-not-induct t)))
```

```
(disable lessp-times-cancellation1)
```

```
(lemma lessp-plus-times-proof ()
       (implies (lessp x a)
                (equal (lessp (plus x (times a b))
                              (times a c))
                       (lessp b c)))
       ((enable-theory addition)
        (enable commutativity-of-times
                lessp-times-cancellation1
                lessp-times1
                lessp-times2
                lessp-times3
                times-add1
                times-zero)
        (induct (lessp b c))))


(lemma lessp-plus-times1 (rewrite)
       (and (equal (lessp (plus a (times b c)) b)
                   (and (lessp a b) (zerop c)))
            (equal (lessp (plus a (times c b)) b)
                   (and (lessp a b) (zerop c)))
            (equal (lessp (plus (times c b) a) b)
                   (and (lessp a b) (zerop c)))
            (equal (lessp (plus (times b c) a) b)
                   (and (lessp a b) (zerop c))))
       ((use (lessp-plus-times-proof (a b) (b c) (c 1) (x a)))
        (enable commutativity-of-plus
                commutativity-of-times
                times-1-arg1)
        (do-not-induct t)))
```

```
(lemma lessp-plus-times2 (rewrite)
       (implies (and (not (zerop a))
                     (lessp x a))
                (and (equal (lessp (plus x (times a b))
                                   (times a c))
                            (lessp b c))
                     (equal (lessp (plus x (times b a))
                                   (times a c))
                            (lessp b c))
                     (equal (lessp (plus x (times a b))
                                   (times c a))
                            (lessp b c))
                     (equal (lessp (plus x (times b a))
                                   (times c a))
                            (lessp b c))
                     (equal (lessp (plus (times a b) x)
                                   (times a c))
                            (lessp b c))
                     (equal (lessp (plus (times b a) x)
                                   (times a c))
                            (lessp b c))
                     (equal (lessp (plus (times a b) x)
                                   (times c a))
                            (lessp b c))
                     (equal (lessp (plus (times b a) x)
                                   (times c a))
                            (lessp b c))))
       ((enable commutativity-of-plus
                commutativity-of-times)
        (use (lessp-plus-times-proof (a a) (b b) (c c) (x x)))
        (do-not-induct t)))


(lemma lessp-1-times (rewrite)
       (equal (lessp 1 (times a b))
              (not (or (zerop a)
                       (zerop b)
                       (and (equal a 1) (equal b 1))))))




;;; meta lemmas to cancel lessp-times and equal-times expressions
;; examples
;; (lessp (times b (times d a)) (times b (times e (times a f)))) ->
;;                                        (and (and (not (zerop a))
;;                                                  (not (zerop b)))
;;                                             (lessp (fix d) (times e f)))
;;
;; (equal (times b (times c d)) (times b d))    ->
;;                                       (or (or (zerop b) (zerop d))
;;                                           (equal (fix c) 1))
```

```
(defn times-tree (x)
  (if (nlistp x)
      ''1
      (if (nlistp (cdr x))
          (list 'fix (car x))
          (if (nlistp (cddr x))
              (list 'times (car x) (cadr x))
              (list 'times (car x) (times-tree (cdr x)))))))


(defn times-fringe (x)
  (if (and (listp x)
           (equal (car x) 'times))
      (append (times-fringe (cadr x)) (times-fringe (caddr x)))
      (cons x nil)))


(defn or-zerop-tree (x)
  (if (nlistp x)
      '(false)
      (if (nlistp (cdr x))
          (list 'zerop (car x))
          (if (nlistp (cddr x))
              (list 'or (list 'zerop (car x)) (list 'zerop (cadr x)))
              (list 'or (list 'zerop (car x)) (or-zerop-tree (cdr x)))))))


(defn and-not-zerop-tree (x)
  (if (nlistp x)
      '(true)
      (if (nlistp (cdr x))
          (list 'not (list 'zerop (car x)))
          (list 'and (list 'not (list 'zerop (car x)))
                (and-not-zerop-tree (cdr x))))))


(lemma numberp-eval$-times (rewrite)
       (implies (equal (car x) 'times)
                (numberp (eval$ t x a))))


(disable numberp-eval$-times)



(lemma eval$-times (rewrite)
       (implies (equal (car x) 'times)
                (equal (eval$ t x a)
                       (times (eval$ t (cadr x) a) (eval$ t (caddr x) a)))))


(disable eval$-times)
```

```
(lemma eval$-or (rewrite)
       (implies (equal (car x) 'or)
                (equal (eval$ t x a)
                       (or (eval$ t (cadr x) a) (eval$ t (caddr x) a)))))

(disable eval$-or)

(lemma eval$-equal (rewrite)
       (implies (equal (car x) 'equal)
                (equal (eval$ t x a)
                       (equal (eval$ t (cadr x) a) (eval$ t (caddr x) a)))))

(disable eval$-equal)

(lemma eval$-lessp (rewrite)
       (implies (equal (car x) 'lessp)
                (equal (eval$ t x a)
                       (lessp (eval$ t (cadr x) a) (eval$ t (caddr x) a)))))

(disable eval$-lessp)

(lemma eval$-quotient (rewrite)
       (implies (equal (car x) 'quotient)
                (equal (eval$ t x a)
                       (quotient (eval$ t (cadr x) a)
                                 (eval$ t (caddr x) a)))))

(disable eval$-quotient)

(lemma eval$-if (rewrite)
       (implies (equal (car x) 'if)
                (equal (eval$ t x a)
                       (if (eval$ t (cadr x) a)
                           (eval$ t (caddr x) a)
                           (eval$ t (cadddr x) a)))))

(disable eval$-if)

(lemma numberp-eval$-times-tree (rewrite)
       (numberp (eval$ t (times-tree x) a))
       ((enable times-tree)))
```

```
(disable numberp-eval$-times-tree)


(lemma lessp-times-arg1 ()
       (implies (not (zerop a))
                (equal (not (lessp (times a x) (times a y)))
                       (not (lessp x y))))
       ((induct (plus a x))
        (enable times correctness-of-cancel-lessp-plus)))


(lemma infer-equality-from-not-lessp ()
       (implies (and (numberp a)
                     (numberp b))
                (equal (and (not (lessp a b)) (not (lessp b a)))
                       (equal a b))))


(lemma equal-times-arg1 (rewrite)
       (implies (not (zerop a))
                (equal (equal (times a x) (times a y))
                       (equal (fix x) (fix y))))
       ((use (lessp-times-arg1 (a a) (x x) (y y))
             (lessp-times-arg1 (a a) (x y) (y x))
             (infer-equality-from-not-lessp (a (times a x)) (b (times a y))))
        (do-not-induct t)))


(disable equal-times-arg1)


(lemma equal-times-bridge (rewrite)
       (equal (equal (times a b)
                     (times c (times a d)))
              (or (zerop a)
                  (equal (fix b) (times c d))))
       ((enable commutativity-of-times
                commutativity2-of-times
                equal-times-0
                equal-times-arg1
                times-zero)))


(disable equal-times-bridge)
```

```
(lemma eval$-times-member (rewrite)
       (implies (member e x)
                (equal (eval$ t (times-tree x) a)
                       (times (eval$ t e a)
                              (eval$ t
                                     (times-tree (delete e x))
                                     a))))
       ((enable delete times-tree
                COMMUTATIVITY-OF-TIMES
                DELETE-NON-MEMBER
                EQUAL-TIMES-0
                EQUAL-TIMES-BRIDGE
                LISTP-DELETE
                MEMBER-NON-LIST
                TIMES-1-ARG1
                TIMES-ZERO)))

(disable eval$-times-member)

(lemma zerop-makes-times-tree-zero (rewrite)
       (implies (and (not (eval$ t (and-not-zerop-tree x) a))
                     (subbagp x y))
                (equal (eval$ t (times-tree y) a) 0))
       ((enable AND-NOT-ZEROP-TREE
                COMMUTATIVITY-OF-TIMES
                EVAL$-TIMES-MEMBER
                SUBBAGP
                TIMES-TREE
                TIMES-ZERO)))

(disable zerop-makes-times-tree-zero)

(lemma or-zerop-tree-is-not-zerop-tree (rewrite)
       (equal (eval$ t (or-zerop-tree x) a)
              (not (eval$ t (and-not-zerop-tree x) a)))
       ((enable AND-NOT-ZEROP-TREE OR-ZEROP-TREE)))

(disable or-zerop-tree-is-not-zerop-tree)
```

```
(lemma zerop-makes-times-tree-zero2 (rewrite)
       (implies (and (eval$ t (or-zerop-tree x) a)
                     (subbagp x y))
                (equal (eval$ t (times-tree y) a) 0))
       ((use (zerop-makes-times-tree-zero)
             (or-zerop-tree-is-not-zerop-tree))
        (enable OR-ZEROP-TREE
                SUBBAGP
                TIMES-TREE)))


(disable zerop-makes-times-tree-zero2)


(lemma times-tree-append (rewrite)
       (equal (eval$ t (times-tree (append x y)) a)
              (times (eval$ t (times-tree x) a) (eval$ t (times-tree y) a)))
       ((enable append
                ASSOCIATIVITY-OF-TIMES
                COMMUTATIVITY-OF-TIMES
                COMMUTATIVITY2-OF-TIMES
                EQUAL-TIMES-0
                EQUAL-TIMES-ARG1
                EQUAL-TIMES-BRIDGE
                NUMBERP-EVAL$-TIMES-TREE
                TIMES-1-ARG1
                TIMES-TREE
                TIMES-ZERO)))


(disable times-tree-append)


(lemma times-tree-of-times-fringe (rewrite)
       (equal (eval$ t (times-tree (times-fringe x)) a)
              (fix (eval$ t x a)))
       ((enable COMMUTATIVITY-OF-TIMES
                EVAL$-TIMES
                TIMES-FRINGE
                TIMES-TREE
                TIMES-TREE-APPEND
                TIMES-ZERO)
        (induct (times-fringe x))))


(disable times-tree-of-times-fringe)
```

```
(defn cancel-lessp-times (x)
  (if (and (equal (car x) 'lessp)
           (equal (caadr x) 'times)
           (equal (caaddr x) 'times))
      (let ((inboth (bagint (times-fringe (cadr x))
                            (times-fringe (caddr x)))))
        (if (listp inboth)
            (list
             'and
             (and-not-zerop-tree inboth)
             (list 'lessp
                   (times-tree (bagdiff (times-fringe (cadr x)) inboth))
                   (times-tree (bagdiff (times-fringe (caddr x)) inboth))))
            x))
      x))
```

```
(lemma eval$-lessp-times-tree-bagdiff (rewrite)
       (implies (and (subbagp x y)
                     (subbagp x z)
                     (eval$ t (and-not-zerop-tree x) a))
                (equal (lessp (eval$ t (times-tree (bagdiff y x)) a)
                              (eval$ t (times-tree (bagdiff z x)) a))
                       (lessp (eval$ t (times-tree y) a)
                              (eval$ t (times-tree z) a))))
       ((enable AND-NOT-ZEROP-TREE
                BAGDIFF
                EVAL$-TIMES-MEMBER
                LESSP-TIMES-CANCELLATION1
                SUBBAGP
                SUBBAGP-CDR1
                SUBBAGP-CDR2
                TIMES-TREE
                ZEROP-MAKES-TIMES-TREE-ZERO)))
```

```
(disable eval$-lessp-times-tree-bagdiff)
```

```
(lemma zerop-makes-lessp-false-bridge (rewrite)
       (implies (and (equal (car x) 'times)
                     (equal (car y) 'times)
                     (not (eval$ t (and-not-zerop-tree
                                    (bagint (times-fringe x)
                                            (times-fringe y)))
                                 a)))
                (equal (lessp (times (eval$ t (cadr x) a)
                                     (eval$ t (caddr x) a))
                              (times (eval$ t (cadr y) a)
                                     (eval$ t (caddr y) a)))
                       f))
       ((enable AND-NOT-ZEROP-TREE
                BAGINT
                COMMUTATIVITY-OF-TIMES
                DELETE
                EQUAL-TIMES-0
                EVAL$-TIMES
                ;MEMBER-CONS
                 ;MEMBER-NON-LIST
                SUBBAGP-BAGINT1
                SUBBAGP-BAGINT2
                TIMES-FRINGE
                TIMES-TREE
                TIMES-TREE-APPEND
                TIMES-TREE-OF-TIMES-FRINGE
                TIMES-ZERO)
        (use (zerop-makes-times-tree-zero (x (bagint (times-fringe x)
                                                     (times-fringe y)))
                                          (y (times-fringe x)))
             (zerop-makes-times-tree-zero (x (bagint (times-fringe x)
                                                     (times-fringe y)))
                                          (y (times-fringe y))))))

(disable zerop-makes-lessp-false-bridge)

(lemma correctness-of-cancel-lessp-times ((meta lessp))
       (equal (eval$ t x a)
              (eval$ t (cancel-lessp-times x) a))
       ((enable CANCEL-LESSP-TIMES
                EVAL$-LESSP-TIMES-TREE-BAGDIFF
                EVAL$-LESSP
                EVAL$-TIMES
                SUBBAGP-BAGINT1
                SUBBAGP-BAGINT2
                TIMES-TREE-OF-TIMES-FRINGE
                ZEROP-MAKES-LESSP-FALSE-BRIDGE)))
```

```
(defn cancel-equal-times (x)
  (if (and (equal (car x) 'equal)
           (equal (caadr x) 'times)
           (equal (caaddr x) 'times))
      (let ((inboth (bagint (times-fringe (cadr x)) (times-fringe (caddr x)))))
        (if (listp inboth)
            (list
             'or
             (or-zerop-tree inboth)
             (list 'equal
                   (times-tree (bagdiff (times-fringe (cadr x)) inboth))
                   (times-tree (bagdiff (times-fringe (caddr x)) inboth))))
            x))
      x))


(lemma zerop-makes-equal-true-bridge (rewrite)
       (implies (and (equal (car x) 'times)
                     (equal (car y) 'times)
                     (eval$ t (or-zerop-tree (bagint (times-fringe x)
                                                     (times-fringe y)))
                            a))
                (equal
                 (equal (times (eval$ t (cadr x) a)
                               (eval$ t (caddr x) a))
                        (times (eval$ t (cadr y) a)
                               (eval$ t (caddr y) a)))
                 t))
       ((enable BAGINT
                COMMUTATIVITY-OF-TIMES
                DELETE
                EQUAL-TIMES-0
                EVAL$-TIMES
                OR-ZEROP-TREE
                SUBBAGP-BAGINT1
                SUBBAGP-BAGINT2
                TIMES-FRINGE
                TIMES-TREE
                TIMES-TREE-APPEND
                TIMES-TREE-OF-TIMES-FRINGE
                TIMES-ZERO)
        (use (zerop-makes-times-tree-zero2 (x (bagint (times-fringe x)
                                                      (times-fringe y)))
                                           (y (times-fringe x)))
             (zerop-makes-times-tree-zero2 (x (bagint (times-fringe x)
                                                      (times-fringe y)))
                                           (y (times-fringe y))))))


(disable zerop-makes-equal-true-bridge)
```

```
(lemma eval$-equal-times-tree-bagdiff (rewrite)
       (implies (and (subbagp x y)
                     (subbagp x z)
                     (not (eval$ t (or-zerop-tree x) a)))
                (equal
                 (equal (eval$ t (times-tree (bagdiff y x)) a)
                        (eval$ t (times-tree (bagdiff z x)) a))
                 (equal (eval$ t (times-tree y) a)
                        (eval$ t (times-tree z) a))))
       ((enable AND-NOT-ZEROP-TREE
                BAGDIFF
                EQUAL-TIMES-ARG1
                EVAL$-TIMES-MEMBER
                NUMBERP-EVAL$-TIMES-TREE
                OR-ZEROP-TREE
                OR-ZEROP-TREE-IS-NOT-ZEROP-TREE
                SUBBAGP
                SUBBAGP-CDR1
                SUBBAGP-CDR2
                TIMES-TREE
                ZEROP-MAKES-TIMES-TREE-ZERO)))


(disable eval$-equal-times-tree-bagdiff)


(lemma cancel-equal-times-preserves-inequality (rewrite)
       (implies (and (subbagp z x)
                     (subbagp z y)
                     (not (equal
                           (eval$ t (times-tree x) a)
                           (eval$ t (times-tree y) a))))
                (not (equal
                      (eval$ t (times-tree (bagdiff x z)) a)
                      (eval$ t (times-tree (bagdiff y z)) a))))
       ((enable BAGDIFF
                EVAL$-TIMES-MEMBER
                SUBBAGP
                SUBBAGP-CDR2
                TIMES-TREE)))


(disable cancel-equal-times-preserves-inequality)
```

```
(lemma cancel-equal-times-preserves-inequality-bridge (rewrite)
  (implies (and (equal (car x) 'times)
                (equal (car y) 'times)
                (not (equal
                      (times (eval$ t (cadr x) a) (eval$ t (caddr x) a))
                      (times (eval$ t (cadr y) a) (eval$ t (caddr y) a)))))
           (not (equal
                 (eval$ t (times-tree (bagdiff (times-fringe x)
                                               (bagint (times-fringe x)
                                                       (times-fringe y))))
                        a)
                 (eval$ t (times-tree (bagdiff (times-fringe y)
                                               (bagint (times-fringe x)
                                                       (times-fringe y))))
                        a))))
  ((enable BAGDIFF BAGINT
           COMMUTATIVITY-OF-TIMES
           SUBBAGP-BAGINT1
           SUBBAGP-BAGINT2
           TIMES-FRINGE
           TIMES-TREE
           TIMES-TREE-APPEND
           TIMES-TREE-OF-TIMES-FRINGE
           TIMES-ZERO)
   (use (cancel-equal-times-preserves-inequality
         (z (bagint (times-fringe x) (times-fringe y)))
         (x (times-fringe x))
         (y (times-fringe y))))))

(disable cancel-equal-times-preserves-inequality-bridge)
```

```
(lemma correctness-of-cancel-equal-times ((meta equal))
       (equal (eval$ t x a)
              (eval$ t (cancel-equal-times x) a))
       ((enable CANCEL-EQUAL-TIMES
                CANCEL-EQUAL-TIMES-PRESERVES-INEQUALITY-BRIDGE
                EVAL$-EQUAL
                EVAL$-EQUAL-TIMES-TREE-BAGDIFF
                EVAL$-TIMES
                SUBBAGP-BAGINT1
                SUBBAGP-BAGINT2
                TIMES-TREE-OF-TIMES-FRINGE
                ZEROP-MAKES-EQUAL-TRUE-BRIDGE)))
```

```
; Define the available  theory of  multiplication.   To  get the list of
; events to put in the theory,  evaluate the following  form in NQTHM at
; this point  in  the  script. This   form  lists all lemmas   which are
; globally enabled, and which have non-null lemma type.
;
; (remove-if-not
;   (function (lambda (x)
;                    (and (member x (lemmas))
;                         (not (assoc x disabled-lemmas))
;                         (not (null (nth 2 (get x 'event))))
;                         (not (member x (nth 2 (get 'addition 'event)))))))
;   chronology)
```

```
(deftheory multiplication
  (EQUAL-TIMES-0
   EQUAL-TIMES-1
   equal-sub1-0
   TIMES-ZERO
   TIMES-ADD1
   COMMUTATIVITY-OF-TIMES
   TIMES-DISTRIBUTES-OVER-PLUS
   COMMUTATIVITY2-OF-TIMES
   ASSOCIATIVITY-OF-TIMES
   TIMES-DISTRIBUTES-OVER-DIFFERENCE
   TIMES-QUOTIENT
   TIMES-1-ARG1
   LESSP-TIMES1
   LESSP-TIMES2
   lessp-times3
   LESSP-PLUS-TIMES1
   LESSP-PLUS-TIMES2
   LESSP-1-TIMES
   correctness-of-cancel-lessp-times
   correctness-of-cancel-equal-times))

; ---------- REMAINDER ----------


(lemma lessp-remainder (rewrite generalize)
       (equal (lessp (remainder x y) y)
              (not (zerop y))))


(lemma remainder-noop (rewrite)
       (implies (lessp a b)
                (equal (remainder a b)
                       (fix a))))


(lemma remainder-of-non-number (rewrite)
       (implies (not (numberp a))
                (equal (remainder a n)
                       (remainder 0 n))))


(lemma remainder-zero (rewrite)
       (implies (zerop x)
                (equal (remainder y x)
                       (fix y))))
```

```
(lemma plus-remainder-times-quotient (rewrite)
       (equal (plus (remainder x y) (times y (quotient x y)))
              (fix x))
       ((enable commutativity2-of-plus
                commutativity-of-plus
                times-zero
                times-add1
                commutativity-of-times)))


(DISABLE PLUS-REMAINDER-TIMES-QUOTIENT)


(lemma remainder-quotient-elim (elim)
       (implies (and (not (zerop y))
                     (numberp x))
                (equal (plus (remainder x y) (times y (quotient x y)))
                       x))
       ((enable plus-remainder-times-quotient)))

; (lemma remainder-sub1 (rewrite)
;        (implies (and (not (zerop a))
;                      (not (zerop b)))
;                 (equal (remainder (sub1 a) b)
;                        (if (equal (remainder a b) 0)
;                            (sub1 b)
;                            (sub1 (remainder a b)))))
;        ((enable lessp-remainder
;                 remainder-noop
;                 remainder-quotient-elim)
;        (enable-theory addition)
;        (induct (remainder a b))))


(lemma remainder-add1 (rewrite)
       (implies (equal (remainder a b) 0)
                (equal (remainder (add1 a) b)
                       (remainder 1 b)))
       ((enable remainder-noop)
        (enable-theory addition)
        (induct (remainder a b))))


(lemma remainder-plus-proof ()
       (implies (equal (remainder b c) 0)
                (equal (remainder (plus a b) c)
                       (remainder a c)))
       ((enable remainder-noop)
        (enable-theory addition)
        (induct (remainder b c))))
```

```
(lemma remainder-plus (rewrite)
       (implies (equal (remainder a c) 0)
                (and (equal (remainder (plus a b) c)
                            (remainder b c))
                     (equal (remainder (plus b a) c)
                            (remainder b c))
                     (equal (remainder (plus x y a) c)
                            (remainder (plus x y) c))))
       ((use (remainder-plus-proof (a b) (b a) (c c))
             (remainder-plus-proof (a a) (b b) (c c))
             (remainder-plus-proof (b a) (a (plus x y)) (c c)))
        (enable commutativity-of-plus commutativity2-of-plus
                associativity-of-plus)))


(lemma equal-remainder-plus-0-proof ()
       (implies (equal (remainder a c) 0)
                (equal (equal (remainder (plus a b) c) 0)
                       (equal (remainder b c) 0)))
       ((enable remainder-plus)))


(lemma equal-remainder-plus-0 (rewrite)
       (implies (equal (remainder a c) 0)
                (and (equal (equal (remainder (plus a b) c) 0)
                            (equal (remainder b c) 0))
                     (equal (equal (remainder (plus b a) c) 0)
                            (equal (remainder b c) 0))
                     (equal (equal (remainder (plus x y a) c) 0)
                            (equal (remainder (plus x y) c) 0))))
       ((use (equal-remainder-plus-0-proof (a a) (b b) (c c))
             (equal-remainder-plus-0-proof (a b) (b a) (c c))
             (equal-remainder-plus-0-proof (a a) (b (plus x y)) (c c)))
        (enable associativity-of-plus commutativity-of-plus
                commutativity2-of-plus)
        (do-not-induct t)))


(lemma equal-remainder-plus-remainder-proof ()
       (implies (lessp a c)
                (equal (equal (remainder (plus a b) c)
                              (remainder b c))
                       (zerop a)))
       ((enable remainder-noop)
        (enable-theory addition)
        (induct (remainder b c))))
```

```
(lemma equal-remainder-plus-remainder (rewrite)
       (implies (lessp a c)
                (and (equal (equal (remainder (plus a b) c)
                                   (remainder b c))
                            (zerop a))
                     (equal (equal (remainder (plus b a) c)
                                   (remainder b c))
                            (zerop a))))
       ((use (equal-remainder-plus-remainder-proof (a a) (b b) (c c)))
        (enable commutativity-of-plus)
        (do-not-induct t)))


(DISABLE EQUAL-REMAINDER-PLUS-REMAINDER)


(lemma remainder-times1-proof ()
       (implies (equal (remainder b c) 0)
                (equal (remainder (times a b) c) 0))
       ((enable-theory multiplication addition)
        (enable remainder-plus
                remainder-noop
                remainder-zero)))


(lemma remainder-times1 (rewrite)
       (implies (equal (remainder b c) 0)
                (and (equal (remainder (times a b) c) 0)
                     (equal (remainder (times b a) c) 0)))
       ((use (remainder-times1-proof (a a) (b b) (c c))
             (remainder-times1-proof (a b) (b a) (c c)))
        (enable commutativity-of-times)))


(lemma remainder-times1-instance-proof ()
       (equal (remainder (times x y) y) 0)
       ((enable commutativity-of-times
                difference-plus-cancellation
                remainder-zero)
        (induct (times x y))))


(lemma remainder-times1-instance (rewrite)
       (and (equal (remainder (times x y) y) 0)
            (equal (remainder (times x y) x) 0))
       ((use (remainder-times1-instance-proof (x x) (y y))
             (remainder-times1-instance-proof (x y) (y x)))
        (enable commutativity-of-times)))
```

```
(lemma remainder-times-times-proof ()
       (equal (remainder (times x y) (times x z))
              (times x (remainder y z)))
       ((enable-theory addition multiplication)
        (enable remainder-zero)
        (induct (remainder y z))))


(lemma remainder-times-times (rewrite)
       (and (equal (remainder (times x y) (times x z))
                   (times x (remainder y z)))
            (equal (remainder (times x z) (times y z))
                   (times (remainder x y) z)))
       ((use (remainder-times-times-proof (x x) (y y) (z z))
             (remainder-times-times-proof (x z) (y x) (z y)))
        (enable commutativity-of-times)))


(DISABLE REMAINDER-TIMES-TIMES)


(lemma remainder-times2-proof ()
       (implies (equal (remainder a z) 0)
                (equal (remainder a (times z y))
                       (times z (remainder (quotient a z) y))))
       ((enable-theory addition multiplication)
        (enable lessp-remainder
                remainder-noop
                remainder-plus
                remainder-quotient-elim
                remainder-times-times
                remainder-times1-instance
                remainder-zero)
        (do-not-induct t)))


(lemma remainder-times2 (rewrite)
       (implies (equal (remainder a z) 0)
                (and (equal (remainder a (times y z))
                            (times z (remainder (quotient a z) y)))
                     (equal (remainder a (times z y))
                            (times z (remainder (quotient a z) y)))))
       ((use (remainder-times2-proof (a a) (y y) (z z)))
        (enable commutativity-of-times)))


(lemma remainder-times2-instance (rewrite)
       (and (equal (remainder (times x y) (times x z))
                   (times x (remainder y z)))
            (equal (remainder (times x z) (times y z))
                   (times (remainder x y) z)))
       ((enable remainder-times-times)))
```

```
(lemma remainder-difference1 (rewrite)
       (implies (equal (remainder a c)
                       (remainder b c))
                (equal (remainder (difference a b) c)
                       (difference (remainder a c)
                                   (remainder b c))))
       ((enable lessp-remainder
                equal-remainder-plus-remainder
                remainder-plus
                remainder-quotient-elim
                remainder-times1-instance)
        (enable-theory addition)
        (do-not-induct t)))


(defn double-remainder-induction (a b c)
  (if (zerop c)
      0
      (if (lessp a c)
          0
          (if (lessp b c)
              0
              (double-remainder-induction (difference a c)
                                          (difference b c)
                                          c)))))


(lemma remainder-difference2 (rewrite)
       (implies (and (equal (remainder a c) 0)
                     (not  (equal (remainder b c) 0)))
                (equal (remainder (difference a b) c)
                       (if (lessp b a)
                           (difference c (remainder b c))
                           0)))
       ((enable equal-remainder-plus-0
                lessp-remainder
                remainder-noop
                remainder-of-non-number
                remainder-quotient-elim
                remainder-times1-instance
                remainder-zero)
        (disable times-distributes-over-plus)
        (enable-theory addition multiplication)
        (induct (double-remainder-induction a b c))))
```

```
(lemma remainder-difference3 (rewrite)
       (implies (and (equal (remainder b c) 0)
                     (not (equal (remainder a c) 0)))
                (equal (remainder (difference a b) c)
                       (if (lessp b a)
                           (remainder a c)
                           0)))
       ((enable remainder-noop
                remainder-of-non-number
                remainder-zero)
        (enable-theory addition)
        (induct (double-remainder-induction a b c))))


(DISABLE REMAINDER-DIFFERENCE3)


(lemma equal-remainder-difference-0 (rewrite)
       (equal (equal (remainder (difference a b) c) 0)
              (if (leq b a)
                  (equal (remainder a c) (remainder b c))
                  t))
       ((enable lessp-remainder
                remainder-difference1
                remainder-of-non-number
                remainder-plus
                remainder-quotient-elim
                remainder-times1-instance
                remainder-zero)
        (enable-theory addition)
        (do-not-induct t)))


(DISABLE EQUAL-REMAINDER-DIFFERENCE-0)


(lemma lessp-plus-fact (rewrite)
       (implies (and (equal (remainder b x) 0)
                     (equal (remainder c x) 0)
                     (lessp b c)
                     (lessp a x))
                (equal (lessp (plus a b) c) t))
       ((enable-theory addition)
        (induct (double-remainder-induction b c x))))


(DISABLE LESSP-PLUS-FACT)
```

```
(lemma remainder-plus-fact ()
       (implies (and (equal (remainder b x) 0)
                     (equal (remainder c x) 0)
                     (lessp a x))
                (equal (remainder (plus a b) c)
                       (plus a (remainder b c))))
       ((enable lessp-plus-fact remainder-noop remainder-difference1)
        (enable-theory addition multiplication)
        (induct (remainder b c))))


(lemma remainder-plus-times-times-proof ()
       (implies (lessp a b)
                (equal (remainder (plus a (times b c)) (times b d))
                       (plus a (remainder (times b c) (times b d)))))
       ((use (remainder-plus-fact (a a) (x b) (b (times b c)) (c (times b d))))
        (enable remainder-times1-instance remainder-times2-instance)
        (do-not-induct t)))


(lemma remainder-plus-times-times (rewrite)
       (implies (lessp a b)
                (and (equal (remainder (plus a (times b c)) (times b d))
                            (plus a (remainder (times b c) (times b d))))
                     (equal (remainder (plus a (times c b)) (times d b))
                            (plus a (remainder (times c b) (times d b))))))
       ((use (remainder-plus-times-times-proof (a a) (b b) (c c) (d d)))
        (enable commutativity-of-times)
        (do-not-induct t)))

; REMAINDER-PLUS-TIMES-TIMES-INSTANCE is the completion of the rules
; TIMES-DISTRIBUTES-OVER-PLUS, REMAINDER-TIMES-TIMES and
; REMAINDER-PLUS-TIMES-TIMES


(lemma remainder-plus-times-times-instance (rewrite)
       (implies (lessp a b)
                (and (equal (remainder (plus a (times b c) (times b d))
                                       (times b e))
                            (plus a (times b (remainder (plus c d) e))))
                     (equal (remainder (plus a (times c b) (times d b))
                                       (times e b))
                            (plus a (times b (remainder (plus c d) e))))))
       ((enable commutativity-of-times
                remainder-times-times
                remainder-plus-times-times)
        (use (times-distributes-over-plus (x b) (y c) (z d)))
        (do-not-induct t)))
```

```
(lemma remainder-remainder (rewrite)
       (implies (equal (remainder b a) 0)
                (equal (remainder (remainder n b) a)
                       (remainder n a)))
       ((induct (remainder n b))
        (enable remainder-plus
                remainder-quotient-elim
                remainder-zero)
        (enable-theory addition multiplication)))


(lemma remainder-1-arg1 (rewrite)
       (equal (remainder 1 x)
              (if (equal x 1) 0 1))
       ((enable difference-leq-arg1)))


(lemma remainder-1-arg2 (rewrite)
       (equal (remainder y 1) 0))


(lemma remainder-x-x (rewrite)
       (equal (remainder x x) 0)
       ((enable equal-difference-0)))


(lemma transitivity-of-divides ()
       (implies (and (equal (remainder a b) 0)
                     (equal (remainder b c) 0))
                (equal (remainder a c) 0))
       ((enable remainder
                remainder-noop
                remainder-plus)
        (enable-theory addition)))


; Define the available  theory of  remainder.   To  get the list of
; events to put in the theory,  evaluate the following  form in NQTHM at
; this point  in  the  script. This   form  lists all lemmas   which are
; globally enabled, and which have non-null lemma type.
;
;
; (let ((lemmas (lemmas)))
;   (remove-if-not
;     (function
;        (lambda (x)
;                (and (member x lemmas)
;                     (not (assoc x disabled-lemmas))
;                     (not (null (nth 2 (get x 'event))))
;                     (not (member x (nth 2 (get 'addition 'event))))
;                     (not (member x (nth 2 (get 'multiplication 'event)))))))
;     chronology))
```

```
(deftheory remainders
  (LESSP-REMAINDER
   REMAINDER-NOOP
   REMAINDER-OF-NON-NUMBER
   REMAINDER-ZERO
   REMAINDER-QUOTIENT-ELIM
   REMAINDER-ADD1
   REMAINDER-PLUS
   EQUAL-REMAINDER-PLUS-0
   REMAINDER-TIMES1
   REMAINDER-TIMES1-INSTANCE
   REMAINDER-TIMES2
   REMAINDER-TIMES2-INSTANCE
   REMAINDER-DIFFERENCE1
   REMAINDER-DIFFERENCE2
   REMAINDER-PLUS-TIMES-TIMES
   REMAINDER-PLUS-TIMES-TIMES-INSTANCE
   REMAINDER-REMAINDER
   REMAINDER-1-ARG1
   REMAINDER-1-ARG2
   REMAINDER-X-X))

; ---------- QUOTIENT, DIVIDES  ----------


(lemma quotient-noop (rewrite)
       (implies (equal b 1)
                (equal (quotient a b)
                       (fix a))))


(lemma quotient-of-non-number (rewrite)
       (implies (not (numberp a))
                (equal (quotient a n)
                       (quotient 0 n))))


(lemma quotient-zero (rewrite)
       (implies (zerop x)
                (equal (quotient y x)
                       0)))


(lemma quotient-add1 (rewrite)
       (implies (equal (remainder a b) 0)
                (equal (quotient (add1 a) b)
                       (if (equal b 1)
                           (add1 (quotient a b))
                           (quotient a b))))
       ((enable quotient-noop)
        (enable-theory addition)
        (induct (remainder a b))))
```

```
(lemma equal-quotient-0 (rewrite)
       (equal (equal (quotient a b) 0)
              (or (zerop b)
                  (lessp a b)))
       ((induct (quotient a b))))


(lemma quotient-sub1 (rewrite)
       (implies (and (not (zerop a))
                     (not (zerop b)))
                (equal (quotient (sub1 a) b)
                       (if (equal (remainder a b) 0)
                           (sub1 (quotient a b))
                           (quotient a b))))
       ((enable quotient-noop equal-quotient-0)
        (enable-theory addition)
        (induct (remainder a b))))


(lemma quotient-plus-proof ()
       (implies (equal (remainder b c) 0)
                (equal (quotient (plus a b) c)
                       (plus (quotient a c) (quotient b c))))
       ((enable remainder-noop)
        (enable-theory addition)
        (induct (remainder b c))))


(lemma quotient-plus (rewrite)
       (implies (equal (remainder a c) 0)
                (and (equal (quotient (plus a b) c)
                            (plus (quotient a c) (quotient b c)))
                     (equal (quotient (plus b a) c)
                            (plus (quotient a c) (quotient b c)))
                     (equal (quotient (plus x y a) c)
                            (plus (quotient (plus x y) c) (quotient a c)))))
       ((use (quotient-plus-proof (a b) (b a) (c c))
             (quotient-plus-proof (a a) (b b) (c c))
             (quotient-plus-proof (a (plus x y)) (b a) (c c)))
        (enable commutativity-of-plus commutativity2-of-plus
                associativity-of-plus)
        (do-not-induct t)))

; I need QUOTIENT-TIMES-INSTANCE to prove the more general
; QUOTIENT-TIMES, but I want QUOTIENT-TIMES-INSTANCE to be tried first
; (i.e. come after QUOTIENT-TIMES in the event list.) So first, prove
; QUOTIENT-TIMES-INSTANCE-TEMP, then prove QUOTIENT-TIMES, and finally
; give QUOTIENT-TIMES-INSTANCE.
```

```
(lemma quotient-times-instance-temp-proof ()
       (equal (quotient (times y x) y)
              (if (zerop y)
                  0
                  (fix x)))
       ((enable times-zero commutativity-of-times
                difference-plus-cancellation)))


(lemma quotient-times-instance-temp (rewrite)
       (and (equal (quotient (times y x) y)
                   (if (zerop y)
                       0
                       (fix x)))
            (equal (quotient (times x y) y)
                   (if (zerop y)
                       0
                       (fix x))))
       ((use (quotient-times-instance-temp-proof (x x) (y y))
             (quotient-times-instance-temp-proof (x y) (y x)))
        (enable commutativity-of-times)))


(DISABLE QUOTIENT-TIMES-INSTANCE-TEMP)


(lemma quotient-times-proof ()
       (implies (equal (remainder a c) 0)
                (equal (quotient (times a b) c)
                       (times b (quotient a c))))
       ((enable-theory addition multiplication remainders)
        (enable quotient-plus quotient-noop equal-quotient-0
                quotient-times-instance-temp)
        (induct (remainder a c))))


(lemma quotient-times (rewrite)
       (implies (equal (remainder a c) 0)
                (and (equal (quotient (times a b) c)
                            (times b (quotient a c)))
                     (equal (quotient (times b a) c)
                            (times b (quotient a c)))))
       ((enable commutativity-of-times)
        (use (quotient-times-proof (a a) (b b) (c c)))
        (do-not-induct t)))
```

```
(lemma quotient-times-instance (rewrite)
       (and (equal (quotient (times y x) y)
                   (if (zerop y)
                       0
                       (fix x)))
            (equal (quotient (times x y) y)
                   (if (zerop y)
                       0
                       (fix x))))
       ((enable quotient-times-instance-temp)))


(lemma quotient-times-times-proof ()
       (equal (quotient (times x y) (times x z))
              (if (zerop x)
                  0
                  (quotient y z)))
       ((enable-theory addition)
        (enable lessp-times-cancellation1 equal-times-0 times-zero
                commutativity-of-times times-distributes-over-difference)
        (induct (quotient y z))))


(lemma quotient-times-times (rewrite)
       (and (equal (quotient (times x y) (times x z))
                   (if (zerop x)
                       0
                       (quotient y z)))
            (equal (quotient (times x z) (times y z))
                   (if (zerop z)
                       0
                       (quotient x y))))
       ((use (quotient-times-times-proof (x x) (y y) (z z))
             (quotient-times-times-proof (x z) (y x) (z y)))
        (enable commutativity-of-times)))


(disable quotient-times-times)


(lemma quotient-difference1 (rewrite)
       (implies (equal (remainder a c) (remainder b c))
                (equal (quotient (difference a b) c)
                       (difference (quotient a c) (quotient b c))))
       ((enable-theory addition multiplication remainders)
        (enable quotient-plus
                quotient-times-instance
                equal-remainder-plus-remainder)
        (do-not-induct t)))
```

```
(lemma quotient-lessp-arg1 (rewrite)
       (implies (lessp a b)
                (equal (quotient a b) 0)))


(lemma quotient-difference2 (rewrite)
       (implies (and (equal (remainder a c) 0)
                     (not (equal (remainder b c) 0)))
                (equal (quotient (difference a b) c)
                       (if (lessp b a)
                           (difference (quotient a c)
                                       (add1 (quotient b c)))
                           0)))
       ((enable equal-quotient-0
                equal-remainder-plus-0
                quotient-times-instance
                quotient-zero)
        (disable times-distributes-over-plus
                 equal-remainder-difference-0
                 remainder-difference3)
        (enable-theory addition multiplication remainders)
        (induct (double-remainder-induction a b c))))


(lemma quotient-difference3 (rewrite)
       (implies (and (equal (remainder b c) 0)
                     (not (equal (remainder a c) 0)))
                (equal (quotient (difference a b) c)
                       (if (lessp b a)
                           (difference (quotient a c)
                                       (quotient b c))
                           0)))
       ((enable equal-quotient-0
                equal-remainder-plus-0
                quotient-lessp-arg1
                quotient-times-instance
                quotient-zero)
        (disable times-distributes-over-plus
                 equal-remainder-difference-0
                 remainder-difference3)
        (enable-theory addition multiplication remainders)
        (induct (double-remainder-induction a b c))))


(lemma remainder-equals-its-first-argument (rewrite)
       (equal (equal a (remainder a b))
              (and (numberp a)
                   (or (zerop b)
                       (lessp a b))))
       ((induct (remainder a b))
        (enable lessp-remainder
                remainder-noop
                remainder-zero)))
```

```
(DISABLE REMAINDER-EQUALS-ITS-FIRST-ARGUMENT)


(lemma quotient-remainder-times (rewrite)
       (equal (quotient (remainder x (times a b)) a)
              (remainder (quotient x a) b))
       ((enable-theory addition multiplication remainders)
        (enable ;lessp-plus-times2
                remainder-equals-its-first-argument
                quotient-noop
                quotient-plus
                quotient-times-instance
                quotient-zero)
        (do-not-induct t)))


(lemma quotient-remainder (rewrite)
       (implies (equal (remainder c a) 0)
                (equal (quotient (remainder b c) a)
                       (remainder (quotient b a) (quotient c a))))
       ((enable-theory addition multiplication remainders)
        (enable quotient-noop
                quotient-plus
                quotient-remainder-times
                quotient-times-instance
                quotient-zero)
        (do-not-induct t)))


(lemma quotient-remainder-instance (rewrite)
       (equal (quotient (remainder x (times a b)) a)
              (remainder (quotient x a) b))
       ((enable quotient-remainder
                quotient-times-instance
                remainder-times1-instance)
        (do-not-induct t)))


(lemma quotient-plus-fact ()
       (implies (and (equal (remainder b x) 0)
                     (equal (remainder c x) 0)
                     (lessp a x))
                (equal (quotient (plus a b) c)
                       (quotient b c)))
       ((enable quotient-lessp-arg1 lessp-plus-fact)
        (enable-theory addition multiplication remainders)
        (induct (quotient b c))))
```

```
(lemma quotient-plus-times-times-proof ()
       (implies (lessp a b)
                (equal (quotient (plus a (times b c)) (times b d))
                       (quotient (times b c) (times b d))))
       ((use (quotient-plus-fact (a a) (x b) (b (times b c)) (c (times b d))))
        (enable remainder-times1-instance)
        (do-not-induct t)))


(lemma quotient-plus-times-times (rewrite)
       (implies (lessp a b)
                (and (equal (quotient (plus a (times b c)) (times b d))
                            (quotient (times b c) (times b d)))
                     (equal (quotient (plus a (times b c)) (times b d))
                            (quotient (times b c) (times b d)))))
       ((use (quotient-plus-times-times-proof (a a) (b b) (c c) (d d)))
        (enable commutativity-of-times)
        (do-not-induct t)))

; QUOTIENT-PLUS-TIMES-TIMES-INSTANCE is the completion of the rules
; QUOTIENT-TIMES-TIMES, QUOTIENT-PLUS-TIMES-TIMES and
; TIMES-DISTRIBUTES-OVER-PLUS


(lemma quotient-plus-times-times-instance (rewrite)
       (implies (lessp a b)
                (and (equal (quotient (plus a (times b c) (times b d))
                                      (times b e))
                            (if (zerop b)
                                0
                                (quotient (plus c d) e)))
                     (equal (quotient (plus a (times c b) (times d b))
                                      (times e b))
                            (if (zerop b)
                                0
                                (quotient (plus d c) e)))))
       ((enable commutativity-of-times
                commutativity-of-plus
                quotient-times-times
                quotient-plus-times-times)
        (use (times-distributes-over-plus (x b) (y c) (z d)))
        (do-not-induct t)))
```

```
(lemma quotient-quotient (rewrite)
       (equal (quotient (quotient b a) c)
              (quotient b (times a c)))
       ((enable-theory addition multiplication remainders)
        (disable times-distributes-over-plus)
        (enable quotient-lessp-arg1
                quotient-plus
                quotient-plus-times-times
                quotient-times-instance
                quotient-times-times
                quotient-noop
                quotient-zero)
        (do-not-induct t)))


(lemma leq-quotient ()
       (implies (lessp a b)
                (leq (quotient a c) (quotient b c)))
       ((induct (double-remainder-induction a b c))
        (enable quotient-lessp-arg1
                quotient-zero)))


(lemma quotient-1-arg2 (rewrite)
       (equal (quotient n 1)
              (fix n)))


(lemma quotient-1-arg1-casesplit ()
       (or (zerop n)
           (equal n 1)
           (lessp 1 n)))


(lemma quotient-1-arg1 (rewrite)
       (equal (quotient 1 n)
              (if (equal n 1)
                  1
                  0))
       ((enable quotient-lessp-arg1)
        (use (quotient-1-arg1-casesplit))))


(lemma quotient-x-x (rewrite)
       (implies (not (zerop x))
                (equal (quotient x x) 1))
       ((enable difference-x-x)))
```

```
(lemma lessp-quotient (rewrite)
       (equal (lessp (quotient i j) i)
              (and (not (zerop i))
                   (not (equal j 1)))))


;; Metalemma to cancel quotient-times expressions

;; ex.
;; (quotient (times a b) (times c (times d a)))  ->
;;                                             (if (not (zerop a))
;;                                                 (quotient (fix b) (times c d))
;;                                              (zero))
;;


(defn cancel-quotient-times (x)
  (if (and (equal (car x) 'quotient)
           (equal (caadr x) 'times)
           (equal (caaddr x) 'times))
      (let ((inboth (bagint (times-fringe (cadr x))
                            (times-fringe (caddr x)))))
        (if (listp inboth)
            (list 'if
                  (and-not-zerop-tree inboth)
                  (list 'quotient
                        (times-tree (bagdiff (times-fringe (cadr x)) inboth))
                        (times-tree (bagdiff (times-fringe (caddr x))
                                             inboth)))
                  '(zero))
            x))
      x))
```

```
(lemma zerop-makes-quotient-zero-bridge (rewrite)
       (implies (and (equal (car x) 'times)
                     (equal (car y) 'times)
                     (not (eval$ t (and-not-zerop-tree
                                    (bagint (times-fringe x)
                                            (times-fringe y)))
                                 a)))
                (equal (quotient (times (eval$ t (cadr x) a)
                                        (eval$ t (caddr x) a))
                                 (times (eval$ t (cadr y) a)
                                        (eval$ t (caddr y) a)))
                       0))
       ((use (zerop-makes-times-tree-zero (x (bagint (times-fringe x)
                                                     (times-fringe y)))
                                          (y (times-fringe x)))
             (zerop-makes-times-tree-zero (x (bagint (times-fringe x)
                                                     (times-fringe y)))
                                          (y (times-fringe y))))
        (enable AND-NOT-ZEROP-TREE
                BAGINT
                DELETE
                EQUAL-QUOTIENT-0
                EQUAL-TIMES-0
                EVAL$-TIMES
                ;MEMBER-CONS
                 ;MEMBER-NON-LIST
                SUBBAGP-BAGINT1
                SUBBAGP-BAGINT2
                TIMES-FRINGE
                TIMES-TREE
                TIMES-TREE-APPEND
                TIMES-TREE-OF-TIMES-FRINGE
                ZEROP-MAKES-LESSP-FALSE-BRIDGE)))

(disable zerop-makes-quotient-zero-bridge)
```

```
(lemma eval$-quotient-times-tree-bagdiff (rewrite)
       (implies (and (subbagp x y)
                     (subbagp x z)
                     (eval$ t (and-not-zerop-tree x) a))
                (equal (quotient (eval$ t (times-tree (bagdiff y x)) a)
                                 (eval$ t (times-tree (bagdiff z x)) a))
                       (quotient (eval$ t (times-tree y) a)
                                 (eval$ t (times-tree z) a))))
       ((enable AND-NOT-ZEROP-TREE
                BAGDIFF
                EQUAL-QUOTIENT-0
                EVAL$-TIMES-MEMBER
                NUMBERP-EVAL$-TIMES-TREE
                QUOTIENT-TIMES-TIMES
                SUBBAGP
                SUBBAGP-CDR1
                SUBBAGP-CDR2
                TIMES-TREE
                ZEROP-MAKES-TIMES-TREE-ZERO)))

(disable eval$-quotient-times-tree-bagdiff)
```

```
(lemma correctness-of-cancel-quotient-times ((meta quotient))
       (equal (eval$ t x a)
              (eval$ t (cancel-quotient-times x) a))
       ((enable CANCEL-QUOTIENT-TIMES
                EVAL$-QUOTIENT-TIMES-TREE-BAGDIFF
                EVAL$-QUOTIENT
                EVAL$-TIMES
                SUBBAGP-BAGINT1
                SUBBAGP-BAGINT2
                TIMES-TREE-OF-TIMES-FRINGE
                ZEROP-MAKES-QUOTIENT-ZERO-BRIDGE)))

; Define the available theory of quotient.  To get the list of events to
; put in the theory, evaluate the following form in NQTHM at  this point
; in the script. This form lists all  lemmas which are globally enabled,
; and which have non-null lemma type.
;
;
; (let ((lemmas (lemmas)))
;   (remove-if-not
;      (function
;        (lambda (x)
;                (and (member x lemmas)
;                     (not (assoc x disabled-lemmas))
;                     (not (null (nth 2 (get x 'event))))
;                     (not (member x (nth 2 (get 'addition 'event))))
;                     (not (member x (nth 2 (get 'multiplication 'event))))
;                     (not (member x (nth 2 (get 'remainders 'event)))))))
;      chronology))
```

```
(deftheory quotients
  (QUOTIENT-NOOP
   QUOTIENT-OF-NON-NUMBER
   QUOTIENT-ZERO
   QUOTIENT-ADD1
   EQUAL-QUOTIENT-0
   QUOTIENT-SUB1
   QUOTIENT-PLUS
   QUOTIENT-TIMES
   QUOTIENT-TIMES-INSTANCE
   QUOTIENT-DIFFERENCE1
   QUOTIENT-LESSP-ARG1
   QUOTIENT-DIFFERENCE2
   QUOTIENT-DIFFERENCE3
   QUOTIENT-REMAINDER-TIMES
   QUOTIENT-REMAINDER
   QUOTIENT-REMAINDER-INSTANCE
   QUOTIENT-PLUS-TIMES-TIMES
   QUOTIENT-PLUS-TIMES-TIMES-INSTANCE
   QUOTIENT-QUOTIENT
   QUOTIENT-1-ARG2
   QUOTIENT-1-ARG1
   QUOTIENT-X-X
   LESSP-QUOTIENT
   correctness-of-cancel-quotient-times))


;;; exp, log, and gcd


(defn exp (i j)
  (if (zerop j)
      1
      (times i (exp i (sub1 j)))))



(defn log (base n)
  (if (lessp base 2)
      0
      (if (zerop n)
          0
          (add1 (log base (quotient n base))))))
```

```
(defn gcd (x y)
  (if (zerop x)
      (fix y)
      (if (zerop y)
          x
          (if (lessp x y)
              (gcd  x (difference y x))
              (gcd (difference x y) y))))
  ((ord-lessp (cons (add1 x) (fix y)))))
```

```
(lemma remainder-exp (rewrite)
       (implies (not (zerop k))
                (equal (remainder (exp n k) n) 0))
       ((enable exp remainder-times1-instance)))
```

```
(defn double-number-induction (i j)
  (if (zerop i)
      0
      (if (zerop j)
          0
          (double-number-induction (sub1 i) (sub1 j)))))
```

```
(lemma remainder-exp-exp (rewrite)
       (implies (leq i j)
                (equal (remainder (exp a j) (exp a i)) 0))
       ((enable exp
                remainder-1-arg2
                remainder-times2-instance)
        (enable-theory addition multiplication)
        (induct (double-number-induction i j))))
```

```
(lemma quotient-exp (rewrite)
       (implies (not (zerop k))
                (equal (quotient (exp n k) n)
                       (if (zerop n)
                           0
                           (exp n (sub1 k)))))
       ((enable exp quotient-times-instance)))
```

```
(lemma exp-zero (rewrite)
       (implies (zerop k)
                (equal (exp n k) 1))
       ((enable exp)))
```

```
(lemma exp-add1 (rewrite)
       (equal (exp n (add1 k))
              (times n (exp n k)))
       ((enable exp)))


(lemma exp-plus (rewrite)
       (equal (exp i (plus j k))
              (times (exp i j) (exp i k)))
       ((enable exp
                associativity-of-times
                commutativity-of-times)))


(lemma exp-0-arg1 (rewrite)
       (equal (exp 0 k)
              (if (zerop k) 1 0))
       ((enable exp)))


(lemma exp-1-arg1 (rewrite)
       (equal (exp 1 k) 1)
       ((enable exp)))


(lemma exp-0-arg2 (rewrite)
       (equal (exp n 0) 1)
       ((enable exp)))


(lemma exp-times (rewrite)
       (equal (exp (times i j) k)
              (times (exp i k) (exp j k)))
       ((enable exp
                associativity-of-times
                commutativity2-of-times
                exp-zero)))


(lemma exp-exp (rewrite)
       (equal (exp (exp i j) k)
              (exp i (times j k)))
       ((enable exp
                exp-zero
                exp-1-arg1
                exp-plus
                exp-times)))
```

```
(lemma equal-exp-0 (rewrite)
       (equal (equal (exp n k) 0)
              (and (zerop n)
                   (not (zerop k))))
       ((enable exp equal-times-0)
        (induct (exp n k))))


(lemma equal-exp-1 (rewrite)
       (equal (equal (exp n k) 1)
              (if (zerop k)
                  t
                  (equal n 1)))
       ((enable exp times-zero times-add1)))


(lemma exp-difference (rewrite)
       (implies (and (leq c b)
                     (not (zerop a)))
                (equal (exp a (difference b c))
                       (quotient (exp a b) (exp a c))))
       ((enable exp)
        (enable-theory addition multiplication remainders quotients)))


(deftheory exponentiation
  (equal-exp-0
   equal-exp-1
   exp-exp
   exp-add1
   exp-times
   exp-1-arg1
   exp-zero
   exp-0-arg2
   exp-0-arg1
   exp-difference
   exp-plus
   quotient-exp
   remainder-exp-exp
   remainder-exp))



(lemma equal-log-0 (rewrite)
       (equal (equal (log base n) 0)
              (or (lessp base 2)
                  (zerop n)))
       ((enable log)
        (induct (log base n))))
```

```
(lemma log-0 (rewrite)
       (implies (zerop n)
                (equal (log base n) 0))
       ((enable log)))


(lemma log-1 (rewrite)
       (implies (lessp 1 base)
                (equal (log base 1) 1))
       ((enable log)
        (induct (log base n))))


(defn double-log-induction (base a b)
  (if (lessp base 2)
      0
      (if (zerop a)
          0
          (if (zerop b)
              0
              (double-log-induction base (quotient a base)
                                    (quotient b base))))))


(lemma leq-log-log nil
       (implies (leq n m)
                (leq (log c n) (log c m)))
       ((enable log)
        (induct (double-log-induction c n m))
        (use (leq-quotient (a n) (b m) (c c)))))


(lemma log-quotient (rewrite)
       (implies (lessp 1 c)
                (equal (log c (quotient n c))
                       (sub1 (log c n))))
       ((enable log)))


(lemma log-quotient-times-proof ()
       (implies (lessp 1 c)
                (equal (log c (quotient n (times c m)))
                       (sub1 (log c (quotient n m)))))
       ((enable log)
        (enable-theory addition multiplication remainders quotients)))
```

```
(lemma log-quotient-times (rewrite)
       (implies (lessp 1 c)
                (and (equal (log c (quotient n (times c m)))
                            (sub1 (log c (quotient n m))))
                     (equal (log c (quotient n (times m c)))
                            (sub1 (log c (quotient n m))))))
       ((use (log-quotient-times-proof (c c) (n n) (m m)))
        (enable commutativity-of-times)))


(lemma log-quotient-exp (rewrite)
       (implies (lessp 1 c)
                (equal (log c (quotient n (exp c m)))
                       (difference (log c n) m)))
       ((enable exp log log-quotient-times)
        (enable-theory addition multiplication remainders quotients)))


(lemma log-times-proof ()
       (implies (and (lessp 1 c)
                     (not (zerop n)))
                (equal (log c (times c n))
                       (add1 (log c n))))
       ((enable log)
        (enable-theory addition multiplication remainders quotients)))


(lemma log-times (rewrite)
       (implies (and (lessp 1 c)
                     (not (zerop n)))
                (and (equal (log c (times c n))
                            (add1 (log c n)))
                     (equal (log c (times n c))
                            (add1 (log c n)))))
       ((use (log-times-proof (c c) (n n)))
        (enable commutativity-of-times)))


(lemma log-times-exp-proof ()
       (implies (and (lessp 1 c)
                     (not (zerop n)))
                (equal (log c (times n (exp c m)))
                       (plus (log c n) m)))
       ((enable log exp)
        (enable-theory addition multiplication remainders quotients)))
```

```
(lemma log-times-exp (rewrite)
       (implies (and (lessp 1 c)
                     (not (zerop n)))
                (and (equal (log c (times n (exp c m)))
                            (plus (log c n) m))
                     (equal (log c (times (exp c m) n))
                            (plus (log c n) m))))
       ((use (log-times-exp-proof (c c) (n n) (m m)))
        (enable commutativity-of-times)))


(lemma log-exp (rewrite)
       (implies (lessp 1 c)
                (equal (log c (exp c n))
                       (add1 n)))
       ((enable log exp log-1)
        (enable-theory addition multiplication remainders quotients)))


(deftheory logs
  (LOG-EXP
   LOG-TIMES-EXP
   LOG-TIMES
   LOG-QUOTIENT-EXP
   LOG-QUOTIENT-TIMES
   LOG-QUOTIENT
   LOG-1
   LOG-0
   EQUAL-LOG-0 EXP-EXP))




(lemma commutativity-of-gcd (rewrite)
       (equal (gcd b a) (gcd a b))
       ((enable gcd)
        (enable-theory addition)))


(defn single-number-induction (n)
  (if (zerop n)
      0
      (single-number-induction (sub1 n))))


(lemma gcd-0 (rewrite)
       (and (equal (gcd 0 x) (fix x))
            (equal (gcd x 0) (fix x)))
       ((enable gcd)))
```

```
(lemma gcd-1 (rewrite)
       (and (equal (gcd 1 x) 1)
            (equal (gcd x 1) 1))
       ((enable gcd)
        (enable-theory addition)
        (induct (single-number-induction x))))


(lemma equal-gcd-0 (rewrite)
       (equal (equal (gcd a b) 0)
              (and (zerop a)
                   (zerop b)))
       ((enable gcd)
        (enable-theory addition)
        (induct (gcd a b))))


(lemma lessp-gcd (rewrite)
       (implies (not (zerop b))
                (and (equal (lessp b (gcd a b)) f)
                     (equal (lessp b (gcd b a)) f)))
       ((enable gcd commutativity-of-gcd)
        (enable-theory addition)))


(lemma gcd-plus-instance-temp-proof ()
       (equal (gcd a (plus a b))
              (gcd a b))
       ((enable gcd commutativity-of-gcd)
        (enable-theory addition)
        (induct (gcd a b))))


(lemma gcd-plus-instance-temp (rewrite)
       (and (equal (gcd a (plus a b))
                   (gcd a b))
            (equal (gcd a (plus b a))
                   (gcd a b)))
       ((enable commutativity-of-plus)
        (use (gcd-plus-instance-temp-proof (a a) (b b)))
        (do-not-induct t)))


(lemma gcd-plus-proof ()
       (implies (equal (remainder b a) 0)
                (equal (gcd a (plus b c))
                       (gcd a c)))
       ((enable gcd commutativity-of-gcd
                gcd-1
                gcd-plus-instance-temp)
        (enable-theory addition)
        (induct (remainder b a))))
```

```
(lemma gcd-plus (rewrite)
       (implies (equal (remainder b a) 0)
                (and (equal (gcd a (plus b c))
                            (gcd a c))
                     (equal (gcd a (plus c b))
                            (gcd a c))
                     (equal (gcd (plus b c) a)
                            (gcd a c))
                     (equal (gcd (plus c b) a)
                            (gcd a c))))
       ((enable commutativity-of-plus commutativity-of-gcd)
        (use (gcd-plus-proof (a a) (b b) (c c)))
        (do-not-induct t)))


(lemma gcd-plus-instance (rewrite)
       (and (equal (gcd a (plus a b))
                   (gcd a b))
            (equal (gcd a (plus b a))
                   (gcd a b)))
       ((enable gcd-plus-instance-temp)
        (do-not-induct t)))


(lemma remainder-gcd (rewrite)
       (and (equal (remainder a (gcd a b)) 0)
            (equal (remainder b (gcd a b)) 0))
       ((enable gcd)
        (enable-theory addition remainders)))


(lemma distributivity-of-times-over-gcd-proof ()
       (equal (gcd (times x z)
                   (times y z))
              (times z (gcd x y)))
       ((enable gcd
                commutativity-of-gcd
                gcd-0
                gcd-plus)
        (enable-theory addition multiplication remainders)))
```

```
(lemma distributivity-of-times-over-gcd (rewrite)
       (and (equal (gcd (times x z) (times y z))
                   (times z (gcd x y)))
            (equal (gcd (times z x) (times y z))
                   (times z (gcd x y)))
            (equal (gcd (times x z) (times z y))
                   (times z (gcd x y)))
            (equal (gcd (times z x) (times z y))
                   (times z (gcd x y))))
       ((use (distributivity-of-times-over-gcd-proof (x x) (y y) (z z)))
        (enable commutativity-of-times)
        (do-not-induct t)))


(lemma gcd-is-the-greatest nil
       (implies (and (not (zerop x))
                     (not (zerop y))
                     (equal (remainder x z) 0)
                     (equal (remainder y z) 0))
                (leq z (gcd x y)))
       ((enable gcd
                commutativity-of-gcd
                distributivity-of-times-over-gcd
                equal-gcd-0)
        (enable-theory addition multiplication remainders)
        (do-not-induct t)))


(lemma common-divisor-divides-gcd (rewrite)
       (implies (and (equal (remainder x z) 0)
                     (equal (remainder y z) 0))
                (equal (remainder (gcd x y) z) 0))
       ((enable gcd
                commutativity-of-gcd
                distributivity-of-times-over-gcd
                equal-gcd-0)
        (enable-theory addition multiplication remainders)
        (do-not-induct t)))

; We prove ASSOCIATIVITY-OF-GCD and COMMUTATIVITY2-OF-GCD roughly the same way.
; Use GCD-IS-THE-GREATEST twice to show that each side of the equality is
; less than or equal to the other side.


(lemma associativity-of-gcd-zero-case ()
       (implies (or (zerop a) (zerop b) (zerop c))
                (equal (gcd (gcd a b) c)
                       (gcd a (gcd b c))))
       ((enable gcd gcd-0)
        (do-not-induct t)))
```

```
(lemma associativity-of-gcd (rewrite)
 (equal (gcd (gcd a b) c)
        (gcd a (gcd b c)))
 ((enable equal-gcd-0
          remainder-gcd)
  (use (gcd-is-the-greatest (x a) (y (gcd b c)) (z (gcd (gcd a b) c)))
       (gcd-is-the-greatest (x (gcd a b)) (y c) (z (gcd a (gcd b c))))
       (associativity-of-gcd-zero-case (a a) (b b) (c c))
       (transitivity-of-divides (a a) (b (gcd a b)) (c (gcd (gcd a b) c)))
       (transitivity-of-divides (a b) (b (gcd a b)) (c (gcd (gcd a b) c)))
       (transitivity-of-divides (a b) (b (gcd b c)) (c (gcd a (gcd b c))))
       (transitivity-of-divides (a c) (b (gcd b c)) (c (gcd a (gcd b c))))
       (common-divisor-divides-gcd (x b) (y c) (z (gcd (gcd a b) c)))
       (common-divisor-divides-gcd (x a) (y b) (z (gcd a (gcd b c))))
       )
  (do-not-induct t)))


(lemma commutativity2-of-gcd-zero-case ()
       (implies (or (zerop a) (zerop b) (zerop c))
                (equal (gcd b (gcd a c))
                       (gcd a (gcd b c))))
       ((enable gcd gcd-0 commutativity-of-gcd)
        (do-not-induct t)))


(lemma commutativity2-of-gcd (rewrite)
  (equal (gcd b (gcd a c))
         (gcd a (gcd b c)))
  ((enable equal-gcd-0
           remainder-gcd)
   (use (gcd-is-the-greatest (x a) (y (gcd b c)) (z (gcd b (gcd a c))))
        (gcd-is-the-greatest (x b) (y (gcd a c)) (z (gcd a (gcd b c))))
        (commutativity2-of-gcd-zero-case (a a) (b b) (c c))
        (transitivity-of-divides (a a) (b (gcd a c)) (c (gcd b (gcd a c))))
        (transitivity-of-divides (a c) (b (gcd a c)) (c (gcd b (gcd a c))))
        (transitivity-of-divides (a b) (b (gcd b c)) (c (gcd a (gcd b c))))
        (transitivity-of-divides (a c) (b (gcd b c)) (c (gcd a (gcd b c))))
        (common-divisor-divides-gcd (x b) (y c) (z (gcd b (gcd a c))))
        (common-divisor-divides-gcd (x a) (y c) (z (gcd a (gcd b c))))
        )
   (do-not-induct t)))


(lemma gcd-x-x (rewrite)
       (equal (gcd x x)
              (fix x))
       ((enable gcd)
        (enable-theory addition)
        (induct (single-number-induction x))))
```

```
(lemma gcd-idempotence (rewrite)
       (and (equal (gcd x (gcd x y)) (gcd x y))
            (equal (gcd y (gcd x y)) (gcd x y)))
       ((enable gcd gcd-x-x
                gcd-plus
                remainder-gcd
                gcd-1
                commutativity-of-gcd)
        (enable-theory addition)
        (induct (gcd x y))))


(deftheory gcds
  (commutativity2-of-gcd
    associativity-of-gcd
    common-divisor-divides-gcd
    distributivity-of-times-over-gcd
    lessp-gcd
    equal-gcd-0
    gcd-0
    gcd-idempotence
    gcd-x-x
    remainder-gcd
    gcd-plus
    gcd-plus-instance
    gcd-1
    commutativity-of-gcd))



(deftheory naturals
  (addition
   multiplication
   remainders
   quotients
   exponentiation
   logs
   gcds))
```

14.18 "integers.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.


;; By Matt Kaufmann, modified from earlier integer library of Bill
;; Bevier and Matt Wilding.  A few functions (even ILESSP) have
;; been changed, but I expect the functionality of this library to
;; include all the functionality of the old one in most or even all
;; cases.

;; Modified from /local/src/nqthm-libs/integers.events to get ILEQ
;; expressed in terms of ILESSP and IDIFFERENCE in terms of INEG and
;; IPLUS.  There are other changes too.  The highlights are the new
;; metalemmas.

;; I'm going to leave the eval$ rules on that are proved here, and
;; leave eval$ off.

;; My intention is that this library be used in a mode in which ILEQ
;; and IDIFFERENCE are left enabled.  Otherwise, the aforementioned
;; meta lemmas may not be very useful, and also a number of additional
;; replacement rules may be needed.

;; There are three theories created by this library.  INTEGER-DEFNS is
;; a list of definitions of all integer functions (not including the
;; cancellation metafunctions and their auxiliaries, though), except
;; that ILEQ and IDIFFERENCE have been omitted.  This is a useful
;; theory for an ENABLE-THEORY hint when one simply wants to blast all
;; integer functions open, and it's also useful if one wants to close
;; them down with a DISABLE-THEORY hint (perhaps to go with an
;; (ENABLE-THEORY T) hint).  Second, ALL-INTEGER-DEFNS is the same as
;; INTEGER-DEFNS except that ILEQ and IDIFFERENCE are included in this
;; one.  Finally, INTEGERS is a list of all events to be "exported as
;; enabled" from this file when working in a mode where everything not
;; enabled by an ENABLE-THEORY hint is to be disabled.  Notice that
;; some rewrite rules have been included that might appear to be
;; unnecessary in light of the metalemmas; that's because metalemmas
;; only work on tame terms.  However, there's no guarantee that the
;; rewrite rules alone will prove very useful (on non-tame terms).
;; Also notice that INTEGER-DEFNS is disjoint from INTEGERS, since we
;; expect the basic definitions (other than ILEQ and IDIFFERENCE) to
;; remain disabled.

;; It's easy to see what I have and haven't placed in INTEGERS, since
;; I'll simply comment out the event names that I want to exclude (see
;; end of this file).

;; One might wish to consider changing (fix-int (minus ...)) in some
;; of the definitions below to (ineg ...).

;; The following meta rules are in this library.
;; (A little documentation added by Matt Wilding July 90)
```

```
;;
;; CORRECTNESS-OF-CANCEL-INEG
;;  cancel the first argument of an iplus term with a member of the second
;;  argument.
;;
;;  ex:  (iplus (ineg y) (iplus (ineg x) (iplus y z)))
;;     -->
;;       (iplus (ineg x) (fix-int z))
;;
;; CORRECTNESS-OF-CANCEL-IPLUS
;;  cancel the sides of an equality of iplus sums
;;
;;  ex:  (equal (iplus x (iplus y z)) (iplus a (iplus z x)))
;;     -->
;;       (equal (fix-int y) (fix-int a))
;;
;; CORRECTNESS-OF-CANCEL-IPLUS-ILESSP
;;  cancel the sides of an ilessp inequality of sums
;;
;;  ex:  (ilessp (iplus x (iplus y z)) (iplus a (iplus z x)))
;;     -->
;;       (ilessp y a)
;;
;; CORRECTNESS-OF-CANCEL-ITIMES
;;  cancel the sides of an equality of itimes products
;;
;;  ex: (equal (itimes x (itimes y z)) (itimes a (itimes z x)))
;;     -->
;;      (if (equal (itimes x z) '0)
;;        t
;;        (equal (fix-int y) (fix-int a)))
;;
;; CORRECTNESS-OF-CANCEL-ITIMES-ILESSP
;;  cancel the sides of an inequality of itimes products
;;
;;  ex: (ilessp (itimes x (itimes y z)) (itimes a (itimes z x)))
;;     -->
;;      (if (ilessp (itimes x z) '0)
;;         (ilessp a y)
;;        (if (ilessp 0 (itimes x z))
;;           (ilessp y a)
;;           f))
;;
;; CORRECTNESS-OF-CANCEL-ITIMES-FACTORS
;;  cancel factors in equality terms
;;  ex: (equal (iplus (itimes x y) x) (itimes z x))
;;     -->
;;      (if (equal (fix-int x) '0)
;;         t
;;        (equal (fix-int (plus y 1)) (fix-int z)))
;;
;; CORRECTNESS-OF-CANCEL-ITIMES-ILESSP-FACTORS
;;  cancel factors in ilessp terms
;;  ex: (equal (iplus (itimes x y) x) (itimes z x))
;;     -->
```

```
;;      (if (ilessp x '0)
;;         (ilessp z (iplus y 1))
;;        (if (ilessp '0 x)
;;           (ilessp (iplus y '1) z)
;;          f))
;;
;; CORRECTNESS-OF-CANCEL-FACTORS-0
;;  factor one side of equality when other side is constant 0
;;
;;  ex: (equal (iplus x (itimes x y)) '0)
;;     -->
;;      (or (equal (fix-int (iplus '1 y)) '0)
;;          (equal (fix-int x) '0))
;;
;; CORRECTNESS-OF-CANCEL-FACTORS-ILESSP-0
;;  factor one side of inequality when other side is constant 0
;;
;;  ex: (ilessp (iplus x (itimes x y)) '0)
;;     -->
;;      (or (and (ilessp (iplus '1 y) '0)
;;               (ilessp '0 x))
;;          (and (ilessp '0 (iplus '1 y))
;;               (ilessp x '0)))
;;
;; CORRECTNESS-OF-CANCEL-INEG-TERMS-FROM-EQUALITY
;;  rewrite equality to remove ineg terms
;;
;;  ex: (equal (iplus (ineg x) (ineg y)) (iplus (ineg z) w))
;;     -->
;;      (equal (fix-int z) (iplus x (iplus y w)))
;;
;; CORRECTNESS-OF-CANCEL-INEG-TERMS-FROM-INEQUALITY
;;  rewrite inequalities to remove ineg terms
;;
;;  ex: (ilessp (iplus (ineg x) (ineg y)) (iplus (ineg z) w))
;;     -->
;;      (ilessp (fix-int z) (iplus x (iplus y w)))


;(note-lib "/local/src/nqthm-libs/naturals")

;(compile-uncompiled-defns "xxx")

; ----------------------------------------------------------------------------
; Integers
; ----------------------------------------------------------------------------

#| The function below has no AND or OR, for efficiency

(defn integerp (x)
  (or (numberp x)
      (and (negativep x)
           (not (zerop (negative-guts x))))))
|#
```

```
(DEFN INTEGERP (X)
  (COND
   ((NUMBERP X) T)
   ((NEGATIVEP X) (NOT (ZEROP (NEGATIVE-GUTS X))))
   (T F)))


(defn fix-int (x)
  (if (integerp x) x 0))

;; Even though I'll include a definition for izerop here, I'll
;; often avoid using it.

(defn izerop (i)
  (equal (fix-int i) 0))

#| old version:

(defn izerop (i)
  (if (integerp i)
      (equal i 0)
      t))
|#


(defn ilessp (i j)
  (if (negativep i)
      (if (negativep j)
          (lessp (negative-guts j) (negative-guts i))
          (if (equal i (minus 0))
              (lessp 0 j)
            t))
      (if (negativep j)
          f
          (lessp i j))))


(defn ileq (i j)
  ;; I expect this to be enabled, in analogy to leq.
  (not (ilessp j i)))
```

```
(defn iplus (x y)
  (if (negativep x)
      (if (negativep y)
          (if (and (zerop (negative-guts x))
                   (zerop (negative-guts y)))
              0
              (minus (plus (negative-guts x)
                           (negative-guts y))))
          (if (lessp y (negative-guts x))
              (minus (difference (negative-guts x) y))
              (difference y (negative-guts x))))
      (if (negativep y)
          (if (lessp x (negative-guts y))
              (minus (difference (negative-guts y) x))
              (difference x (negative-guts y)))
          (plus x y))))


(defn ineg (x)
  (if (negativep x)
      (negative-guts x)
      (if (zerop x)
          0
          (minus x))))


(defn idifference (x y)
  ;; I find it troublesome to reason separately about idifference,
  ;; especially for metalemmas, so I intend to keep it enabled.
  (iplus x (ineg y)))


(defn iabs (i)
  (if (negativep i)
      (negative-guts i)
      (fix i)))


(defn itimes (i j)
  (if (negativep i)
      (if (negativep j)
          (times (negative-guts i) (negative-guts j))
          (fix-int (minus (times (negative-guts i) j))))
      (if (negativep j)
          (fix-int (minus (times i (negative-guts j))))
          (times i j))))
```

```
(defn iquotient (i j)
  (if (equal (fix-int j) 0)
      0
      (if (negativep i)
          (if (negativep j)
              (if (equal (remainder (negative-guts i) (negative-guts j)) 0)
                  (quotient (negative-guts i) (negative-guts j))
                  (add1 (quotient (negative-guts i) (negative-guts j))))
              (if (equal (remainder (negative-guts i) j) 0)
                  (fix-int (minus (quotient (negative-guts i) j)))
                  (fix-int (minus (add1 (quotient (negative-guts i) j))))))
          (if (negativep j)
              (fix-int (minus (quotient i (negative-guts j))))
              (quotient i j)))))
```

```
(defn iremainder (i j)
  (idifference i (itimes j (iquotient i j))))
```

```
(defn idiv (i j)
  (if (equal (fix-int j) 0)
      0
      (if (negativep i)
          (if (negativep j)
              (quotient (negative-guts i) (negative-guts j))
              (if (equal (remainder (negative-guts i) j) 0)
                  (fix-int (minus (quotient (negative-guts i) j)))
                  (fix-int (minus (add1 (quotient (negative-guts i) j))))))
          (if (negativep j)
              (if (equal (remainder i (negative-guts j)) 0)
                  (fix-int (minus (quotient i (negative-guts j))))
                  (fix-int (minus (add1 (quotient i (negative-guts j))))))
              (quotient i j)))))
```

```
(defn imod (i j)
  (idifference (fix-int i) (itimes j (idiv i j))))
```

```
(defn iquo (i j)
  (if (equal (fix-int j) 0)
      0
      (if (negativep i)
          (if (negativep j)
              (quotient (negative-guts i) (negative-guts j))
              (fix-int (minus (quotient (negative-guts i) j))))
          (if (negativep j)
              (fix-int (minus (quotient i (negative-guts j))))
              (quotient i j)))))
```

```
(defn irem (i j)
  (idifference (fix-int i) (itimes j (iquo i j))))

; ---------- DEFTHEORY events for definitions ----------


(deftheory integer-defns
  ;; omits ILEQ and IDIFFERENCE and IZEROP
  (integerp fix-int ilessp iplus ineg iabs itimes
            iquotient iremainder idiv imod iquo irem))


(deftheory all-integer-defns
  (integerp fix-int izerop ilessp ileq iplus ineg idifference iabs itimes
            iquotient iremainder idiv imod iquo irem))


(disable integerp)

(disable fix-int)

(disable ilessp)

(disable iplus)

(disable ineg)

(disable iabs)

(disable itimes)
;; I've disabled the rest later in the file, just because the lemmas
;; about division were (re-)proved with the remaining functions enabled.

; ---------- INTEGERP ----------


(lemma integerp-fix-int (rewrite)
       (integerp (fix-int x))
       ((enable integerp fix-int)))


(lemma integerp-iplus (rewrite)
       (integerp (iplus x y))
       ((enable integerp iplus)))


(lemma integerp-idifference (rewrite)
       (integerp (idifference x y))
       ((enable integerp-iplus idifference)))
```

```
(lemma integerp-ineg (rewrite)
       (integerp (ineg x))
       ((enable integerp ineg)))


(lemma integerp-iabs (rewrite)
       (integerp (iabs x))
       ((enable integerp iabs)))


(lemma integerp-itimes (rewrite)
       (integerp (itimes x y))
       ((enable integerp itimes fix-int)))

; ---------- FIX-INT ----------

;; The first of these, FIX-INT-REMOVER, is potentially dangerous from
;; a backchaining point of view, but I believe it's necessary.  At least
;; the lemmas below it should go a long way toward preventing its application.


(lemma fix-int-remover (rewrite)
  (implies (integerp x)
           (equal (fix-int x) x))
  ((enable fix-int integerp)))


(lemma fix-int-fix-int (rewrite)
       (equal (fix-int (fix-int x))
              (fix-int x))
       ((enable fix-int integerp)))


(lemma fix-int-iplus (rewrite)
       (equal (fix-int (iplus a b))
              (iplus a b))
       ((enable fix-int integerp iplus)))


(lemma fix-int-idifference (rewrite)
       (equal (fix-int (idifference a b))
              (idifference a b))
       ((enable fix-int-iplus idifference)))


(lemma fix-int-ineg (rewrite)
       (equal (fix-int (ineg x))
              (ineg x))
       ((enable fix-int integerp ineg)))
```

```
(lemma fix-int-iabs (rewrite)
       (equal (fix-int (iabs x))
              (iabs x))
       ((enable integerp fix-int iabs)))


(lemma fix-int-itimes (rewrite)
       (equal (fix-int (itimes x y))
              (itimes x y))
       ((enable integerp itimes fix-int)))

; ---------- INEG ----------


(lemma ineg-iplus (rewrite)
       (equal (ineg (iplus a b))
              (iplus (ineg a) (ineg b)))
       ((enable iplus ineg plus-zero-arg2)))


(lemma ineg-ineg (rewrite)
       (equal (ineg (ineg x))
              (fix-int x))
       ((enable ineg fix-int integerp)))


(lemma ineg-fix-int (rewrite)
       (equal (ineg (fix-int x))
              (ineg x))
       ((enable ineg fix-int integerp)))


(lemma ineg-of-non-integerp (rewrite)
  (implies (not (integerp x))
           (equal (ineg x) 0))
  ((enable ineg integerp)))

;; I don't want the backchaining to slow down the prover.

(disable ineg-of-non-integerp)


(lemma ineg-0 (rewrite)
  (equal (ineg 0) 0)
  ((enable ineg)))

; ---------- IPLUS ----------

;; The first two of these really aren't necessary, in light
;; of the cancellation metalemma.
```

```
(lemma iplus-left-id (rewrite)
       (implies (not (integerp x))
                (equal (iplus x y)
                       (fix-int y)))
       ((enable integerp iplus fix-int)))

;; I don't want the backchaining to slow down the prover.

(disable iplus-left-id)


(lemma iplus-right-id (rewrite)
       (implies (not (integerp y))
                (equal (iplus x y)
                       (fix-int x)))
       ((enable integerp iplus fix-int plus-zero-arg2)))

;; I don't want the backchaining to slow down the prover.

(disable iplus-right-id)


(lemma iplus-0-left (rewrite)
       (equal (iplus 0 x) (fix-int x))
       ((enable iplus fix-int integerp)))


(lemma iplus-0-right (rewrite)
       ;; just in case we turn off commutativity
       (equal (iplus x 0) (fix-int x))
       ((enable iplus fix-int integerp)))


(lemma commutativity2-of-iplus (rewrite)
       (equal (iplus x (iplus y z)) (iplus y (iplus x z)))
       ((enable iplus commutativity2-of-plus)))


(lemma commutativity-of-iplus (rewrite)
       (equal (iplus x y) (iplus y x))
       ((enable iplus commutativity2-of-iplus)))


(lemma associativity-of-iplus (rewrite)
       (equal (iplus (iplus x y) z) (iplus x (iplus y z)))
       ((enable iplus)
        (enable-theory addition)))
```

```
(lemma iplus-cancellation-1 (rewrite)
       (equal (equal (iplus a b) (iplus a c))
              (equal (fix-int b) (fix-int c)))
       ((enable iplus fix-int integerp)
        (enable-theory addition)))


(lemma iplus-cancellation-2 (rewrite)
       (equal (equal (iplus b a) (iplus c a))
              (equal (fix-int b) (fix-int c)))
       ((use (iplus-cancellation-1))
        (enable commutativity-of-iplus)))


(lemma iplus-ineg1 (rewrite)
       (equal (iplus (ineg a) a) 0)
       ((enable iplus ineg integerp fix-int)))


(lemma iplus-ineg2 (rewrite)
       (equal (iplus a (ineg a)) 0)
       ((enable iplus ineg integerp fix-int)))


(lemma iplus-fix-int1 (rewrite)
       (equal (iplus (fix-int a) b)
              (iplus a b))
       ((enable iplus fix-int integerp plus-zero-arg2)
        (do-not-induct t)))


(lemma iplus-fix-int2 (rewrite)
       (equal (iplus a (fix-int b))
              (iplus a b))
       ((enable iplus fix-int integerp plus-zero-arg2)
        (do-not-induct t)))

; ---------- IDIFFERENCE ----------

;; mostly omitted, but I'll keep a few


(lemma idifference-fix-int1 (rewrite)
       (equal (idifference (fix-int a) b)
              (idifference a b))
       ((enable idifference iplus-fix-int1)
        (do-not-induct t)))
```

```
(lemma idifference-fix-int2 (rewrite)
       (equal (idifference a (fix-int b))
              (idifference a b))
       ((enable idifference iplus-fix-int2 ineg-fix-int)
        (do-not-induct t)))


; ----------------------------------------------------------------------------
; Cancel INEG
; ----------------------------------------------------------------------------

;; We assume that the given term (IPLUS x y) has the property that y
;; has already been reduced and x is not an iplus-term.  So, the only
;; question is whether or not the formal negative of x appears in the
;; fringe of y.

#| The function below has no AND or OR, for efficiency
```

```
(defn cancel-ineg-aux (x y)
  ;; returns nil or else a new term provably equal to (IPLUS x y)
  (if (and (listp x)
           (equal (car x) 'ineg))
      (cond
       ((equal y (cadr x))
        ''0)
       ((and (listp y)
             (equal (car y) 'iplus))
        (let ((y1 (cadr y)) (y2 (caddr y)))
          (if (equal y1 (cadr x))
              (list 'fix-int y2)
            (let ((z (cancel-ineg-aux x y2)))
              (if z
                  (list 'iplus y1 z)
                f)))))
       (t f))
    (cond
     ((nlistp y)
      f)
     ((equal (car y) 'ineg)
      (if (equal x (cadr y))
          ''0
        f))
     ((equal (car y) 'iplus)
      (let ((y1 (cadr y)) (y2 (caddr y)))
        (if (and (listp y1)
                 (equal (car y1) 'ineg)
                 (equal x (cadr y1)))
            (list 'fix-int y2)
          (let ((z (cancel-ineg-aux x y2)))
            (if z
                (list 'iplus y1 z)
              f)))))
     (t f))))
|#
```

```
(DEFN CANCEL-INEG-AUX (X Y)
      (COND
        ((LISTP X)
         (COND
           ((EQUAL (CAR X) 'INEG)
            (COND
              ((EQUAL Y (CADR X)) ''0)
              ((LISTP Y)
               (IF (EQUAL (CAR Y) 'IPLUS)
                   (COND
                     ((EQUAL (CADR Y) (CADR X))
                      (LIST 'FIX-INT (CADDR Y)))
                     ((CANCEL-INEG-AUX X (CADDR Y))
                      (LIST 'IPLUS (CADR Y)
                            (CANCEL-INEG-AUX X (CADDR Y))))
                     (T F))
                   F))
              (T F)))
           ((NLISTP Y) F)
           ((EQUAL (CAR Y) 'INEG) (IF (EQUAL X (CADR Y)) ''0 F))
           ((EQUAL (CAR Y) 'IPLUS)
            (COND
              ((LISTP (CADR Y))
               (COND
                 ((EQUAL (CAADR Y) 'INEG)
                  (COND
                    ((EQUAL X (CADADR Y)) (LIST 'FIX-INT (CADDR Y)))
                    ((CANCEL-INEG-AUX X (CADDR Y))
                     (LIST 'IPLUS (CADR Y)
                           (CANCEL-INEG-AUX X (CADDR Y))))
                    (T F)))
                 ((CANCEL-INEG-AUX X (CADDR Y))
                  (LIST 'IPLUS (CADR Y) (CANCEL-INEG-AUX X (CADDR Y))))
                 (T F)))
              ((CANCEL-INEG-AUX X (CADDR Y))
               (LIST 'IPLUS (CADR Y) (CANCEL-INEG-AUX X (CADDR Y))))
              (T F)))
           (T F)))
        ((NLISTP Y) F)
        ((EQUAL (CAR Y) 'INEG) (IF (EQUAL X (CADR Y)) ''0 F))
        ((EQUAL (CAR Y) 'IPLUS)
         (COND
           ((LISTP (CADR Y))
            (COND
              ((EQUAL (CAADR Y) 'INEG)
               (COND
                 ((EQUAL X (CADADR Y)) (LIST 'FIX-INT (CADDR Y)))
                 ((CANCEL-INEG-AUX X (CADDR Y))
                  (LIST 'IPLUS (CADR Y) (CANCEL-INEG-AUX X (CADDR Y))))
                 (T F)))
              ((CANCEL-INEG-AUX X (CADDR Y))
               (LIST 'IPLUS (CADR Y) (CANCEL-INEG-AUX X (CADDR Y))))
              (T F)))
           ((CANCEL-INEG-AUX X (CADDR Y))
```

```
            (LIST 'IPLUS (CADR Y) (CANCEL-INEG-AUX X (CADDR Y))))
           (T F)))
        (T F)))

#| The function below has no AND or OR, for efficiency

(defn cancel-ineg (x)
  (if (and (listp x)
           (equal (car x) 'iplus))
      (let ((temp (cancel-ineg-aux (cadr x) (caddr x))))
        (if temp
            temp
          x))
    x))
|#


(DEFN CANCEL-INEG (X)
      (IF (LISTP X)
          (IF (EQUAL (CAR X) 'IPLUS)
              (IF (CANCEL-INEG-AUX (CADR X) (CADDR X))
                  (CANCEL-INEG-AUX (CADR X) (CADDR X))
                  X)
              X)
          X))

;; It seems a big win to turn off eval$.  I'll leave the recursive step out in
;; hopes that rewrite-eval$ handles it OK.


(prove-lemma eval$-list-cons (rewrite)
  (equal (eval$ 'list (cons x y) a)
         (cons (eval$ t x a)
               (eval$ 'list y a))))


(prove-lemma eval$-list-nlistp (rewrite)
  (implies (nlistp x)
           (equal (eval$ 'list x a)
                  nil)))


(prove-lemma eval$-litatom (rewrite)
  (implies (litatom x)
           (equal (eval$ t x a)
                  (cdr (assoc x a)))))

#|
```

```
(prove-lemma eval$-quotep (rewrite)
  (equal (eval$ t (list 'quote x) a)
         x))
|#

;; In place of the above I'll do the following, from
;; the naturals library.

(enable eval$-quote)


(prove-lemma eval$-other (rewrite)
  (implies (and (not (litatom x))
                (nlistp x))
           (equal (eval$ t x a)
                  x)))


(disable eval$)

;; What I'd like to do is say what (eval$ t (cancel-ineg-aux x y) a),
;; but a rewrite rule will loop because of the recursion.  So I
;; introduce a silly auxiliary function so that the opening-up
;; heuristics can help me.  The function body has (listp y) tests
;; so that it can be accepted.
```

```
(defn eval$-cancel-ineg-aux-fn (x y a)
  (if (and (listp x)
           (equal (car x) 'ineg))
      (cond
       ((equal y (cadr x))
        0)
       (t
        (let ((y1 (cadr y)) (y2 (caddr y)))
          (if (equal y1 (cadr x))
              (fix-int (eval$ t y2 a))
            (if (listp y);; silly acceptability thing
                (iplus (eval$ t y1 a)
                       (eval$-cancel-ineg-aux-fn x y2 a))
              0)))))
    (cond
     ((equal (car y) 'ineg)
      0)
     (t
      (let ((y1 (cadr y)) (y2 (caddr y)))
        (if (and (listp y1)
                 (equal (car y1) 'ineg)
                 (equal x (cadr y1)))
            (fix-int (eval$ t y2 a))
          (if (listp y);; silly acceptability thing
              (iplus (eval$ t y1 a)
                     (eval$-cancel-ineg-aux-fn x y2 a))
            0)))))))


(prove-lemma eval$-cancel-ineg-aux-is-its-fn (rewrite)
  (implies (not (equal (cancel-ineg-aux x y) f))
           (equal (eval$ t (cancel-ineg-aux x y) a)
                  (eval$-cancel-ineg-aux-fn x y a))))


(prove-lemma iplus-ineg3 (rewrite)
  (equal (iplus (ineg x) (iplus x y))
         (fix-int y))
  ((enable-theory integer-defns)))


(prove-lemma iplus-ineg4 (rewrite)
  (equal (iplus x (iplus (ineg x) y))
         (fix-int y))
  ((use (iplus-ineg3 (x (ineg x)) (y y)))))


(prove-lemma iplus-ineg-promote (rewrite)
  (equal (iplus y (ineg x))
         (iplus (ineg x) y)))
```

```
(prove-lemma iplus-x-y-ineg-x (rewrite)
  (equal (iplus x (iplus y (ineg x)))
         (fix-int y)))


(disable iplus-ineg-promote)


(prove-lemma correctness-of-cancel-ineg-aux (rewrite)
  (implies (not (equal (cancel-ineg-aux x y) f))
           (equal (eval$-cancel-ineg-aux-fn x y a)
                  (iplus (eval$ t x a) (eval$ t y a))))
  ((induct (cancel-ineg-aux x y))))


(prove-lemma correctness-of-cancel-ineg ((meta iplus))
  (equal (eval$ t x a)
         (eval$ t (cancel-ineg x) a))
  ((disable cancel-ineg-aux)))


(disable correctness-of-cancel-ineg-aux)

; ---------------------------------------------------------------------------
; Cancel IPLUS
; ---------------------------------------------------------------------------

;; All I do here is cancel like terms from both sides.  The problem of
;; handling INEG cancellation IS handled completely separately above.
;; That hasn't always been the case -- in my first try I attempted to
;; integrate the operations.  But now I see that for things like
;; (equal z (iplus x (iplus y (ineg x)))) the integrated approach will
;; fail.  Also, thanks to Matt Wilding, for pointing out that the
;; "four squares" example that Bill Pase sent me ran faster with the
;; newer approach (on his previously-implemented version for the
;; rationals).

#| The function below has no AND or OR, for efficiency

(defn iplus-fringe (x)
  (if (and (listp x)
           (equal (car x)
                  (quote iplus)))
      (append (iplus-fringe (cadr x))
              (iplus-fringe (caddr x)))
      (cons x nil)))
|#
```

```
(DEFN IPLUS-FRINGE (X)
  (IF (LISTP X)
      (IF (EQUAL (CAR X) 'IPLUS)
          (APPEND (IPLUS-FRINGE (CADR X)) (IPLUS-FRINGE (CADDR X)))
          (LIST X))
      (LIST X)))


(prove-lemma lessp-count-listp-cdr (rewrite)
  (implies (listp (cdr x))
           (lessp (count (cdr x)) (count x))))


(defn iplus-tree-rec (l)
  (if (nlistp (cdr l))
      (car l)
    (list (quote iplus)
          (car l)
          (iplus-tree-rec (cdr l)))))


(defn iplus-tree (l)
  (if (listp l)
      (if (listp (cdr l))
          (iplus-tree-rec l)
        (list (quote fix-int)
              (car l)))
    (quote (quote 0))))


(defn iplus-list (x)
  (if (listp x)
      (iplus (car x)
             (iplus-list (cdr x)))
    0))


(prove-lemma integerp-iplus-list (rewrite)
  (integerp (iplus-list x)))


(prove-lemma eval$-iplus-tree-rec (rewrite)
  (equal (eval$ t (iplus-tree-rec x) a)
         (if (listp x)
             (if (listp (cdr x))
                 (iplus-list (eval$ 'list x a))
               (eval$ t (car x) a))
           0)))


(prove-lemma eval$-iplus-tree (rewrite)
  (equal (eval$ t (iplus-tree x) a)
         (iplus-list (eval$ 'list x a))))
```

```
(prove-lemma eval$-list-append (rewrite)
  (equal (eval$ 'list (append x y) a)
         (append (eval$ 'list x a)
                 (eval$ 'list y a))))

#| The function below has no AND or OR, for efficiency

(defn cancel-iplus (x)
  (if (and (listp x)
           (equal (car x) (quote equal)))
      (if (and (listp (cadr x))
               (equal (caadr x) (quote iplus))
               (listp (caddr x))
               (equal (caaddr x) (quote iplus)))
          (let ((xs (iplus-fringe (cadr x)))
                (ys (iplus-fringe (caddr x))))
            (let ((bagint (bagint xs ys)))
              (if (listp bagint)
                  (list (quote equal)
                        (iplus-tree (bagdiff xs bagint))
                        (iplus-tree (bagdiff ys bagint)))
                x)))
        (if (and (listp (cadr x))
                 (equal (caadr x) (quote iplus))
                 ;; We don't want to introduce the IF below unless something
                 ;; is "gained", or else we may get into an infinite
                 ;; rewriting loop.
                 (member (caddr x) (iplus-fringe (cadr x))))
            (list (quote if)
                  (list (quote integerp) (caddr x))
                  (list (quote equal)
                        (iplus-tree (delete (caddr x) (iplus-fringe (cadr x))))
                        ''0)
                  (list (quote quote) f))
          (if (and (listp (caddr x))
                   (equal (caaddr x) (quote iplus))
                   (member (cadr x) (iplus-fringe (caddr x))))
              (list (quote if)
                    (list (quote integerp) (cadr x))
                    (list (quote equal)
                          ''0
                          (iplus-tree (delete (cadr x)
                                              (iplus-fringe (caddr x)))))
                    (list (quote quote) f))
            x)))
    x))
|#
```

```
(DEFN CANCEL-IPLUS (X)
  (IF (LISTP X)
      (IF (EQUAL (CAR X) 'EQUAL)
          (COND
           ((LISTP (CADR X))
            (COND
             ((EQUAL (CAADR X) 'IPLUS)
              (COND
               ((LISTP (CADDR X))
                (COND
                 ((EQUAL (CAADDR X) 'IPLUS)
                  (IF (LISTP (BAGINT (IPLUS-FRINGE (CADR X))
                                     (IPLUS-FRINGE (CADDR X))))
                      (LIST 'EQUAL
                            (IPLUS-TREE
                             (BAGDIFF (IPLUS-FRINGE (CADR X))
                                      (BAGINT (IPLUS-FRINGE (CADR X))
                                              (IPLUS-FRINGE (CADDR X)))))
                            (IPLUS-TREE
                             (BAGDIFF (IPLUS-FRINGE (CADDR X))
                                      (BAGINT (IPLUS-FRINGE (CADR X))
                                              (IPLUS-FRINGE (CADDR X))))))
                      X))
                 ((MEMBER (CADDR X) (IPLUS-FRINGE (CADR X)))
                  (LIST 'IF (LIST 'INTEGERP (CADDR X))
                        (CONS 'EQUAL
                              (CONS
                               (IPLUS-TREE
                                (DELETE (CADDR X)
                                        (IPLUS-FRINGE (CADR X))))
                               '('0)))
                        (LIST 'QUOTE F)))
                 (T X)))
               ((MEMBER (CADDR X) (IPLUS-FRINGE (CADR X)))
                (LIST 'IF (LIST 'INTEGERP (CADDR X))
                      (CONS 'EQUAL
                            (CONS
                             (IPLUS-TREE
                              (DELETE (CADDR X)
                                      (IPLUS-FRINGE (CADR X))))
                             '('0)))
                      (LIST 'QUOTE F)))
               (T X)))
             ((LISTP (CADDR X))
              (IF (EQUAL (CAADDR X) 'IPLUS)
                  (IF (MEMBER (CADR X) (IPLUS-FRINGE (CADDR X)))
                      (LIST 'IF (LIST 'INTEGERP (CADR X))
                            (LIST 'EQUAL ''0
                                  (IPLUS-TREE
                                   (DELETE (CADR X)
                                           (IPLUS-FRINGE (CADDR X)))))
                            (LIST 'QUOTE F))
                      X)
                  X))
```

```
          (T X)))
        ((LISTP (CADDR X))
         (IF (EQUAL (CAADDR X) 'IPLUS)
             (IF (MEMBER (CADR X) (IPLUS-FRINGE (CADDR X)))
                 (LIST 'IF (LIST 'INTEGERP (CADR X))
                       (LIST 'EQUAL ''0
                             (IPLUS-TREE
                              (DELETE (CADR X)
                                      (IPLUS-FRINGE (CADDR X)))))
                       (LIST 'QUOTE F))
                 X)
             X))
        (T X))
       X)
   X))
```

```
(lemma eval$-cancel-iplus (rewrite)
  (equal
   (eval$ t (cancel-iplus x) a)
   (if (and (listp x)
            (equal (car x) (quote equal)))
       (if (and (listp (cadr x))
                (equal (caadr x) (quote iplus))
                (listp (caddr x))
                (equal (caaddr x) (quote iplus)))
           (let ((xs (iplus-fringe (cadr x)))
                 (ys (iplus-fringe (caddr x))))
             (let ((bagint (bagint xs ys)))
               (if (listp bagint)
                   (equal
                    (iplus-list (eval$ 'list (bagdiff xs (bagint xs ys)) a))
                    (iplus-list (eval$ 'list (bagdiff ys (bagint xs ys)) a)))
                 (eval$ t x a))))
         (if (and (listp (cadr x))
                  (equal (caadr x) (quote iplus))
                  (member (caddr x) (iplus-fringe (cadr x))))
             (if (integerp (eval$ t (caddr x) a))
                 (equal (iplus-list
                         (eval$ 'list (delete (caddr x)
                                              (iplus-fringe (cadr x))) a))
                        0)
               f)
           (if (and (listp (caddr x))
                    (equal (caaddr x) (quote iplus))
                    (member (cadr x) (iplus-fringe (caddr x))))
               (if (integerp (eval$ t (cadr x) a))
                   (equal 0
                          (iplus-list
                           (eval$ 'list (delete (cadr x)
                                                (iplus-fringe (caddr x))) a)))
                 f)
             (eval$ t x a))))
     (eval$ t x a)))
  ((enable eval$-iplus-tree cancel-iplus eval$-list-cons
           eval$-litatom eval$-quote)
   (disable eval$)))


(disable cancel-iplus)


(prove-lemma eval$-iplus-list-delete (rewrite)
  (implies (member z y)
           (equal (iplus-list (eval$ 'list (delete z y) a))
                  (idifference (iplus-list (eval$ 'list y a))
                               (eval$ t z a)))))
```

```
(prove-lemma eval$-iplus-list-bagdiff (rewrite)
  (implies (subbagp x y)
           (equal (iplus-list (eval$ 'list (bagdiff y x) a))
                  (idifference (iplus-list (eval$ 'list y a))
                               (iplus-list (eval$ 'list x a))))))


(prove-lemma iplus-list-append (rewrite)
  (equal (iplus-list (append x y))
         (iplus (iplus-list x) (iplus-list y))))


(disable iplus-tree) ;; because we want to use EVAL$-IPLUS-TREE for now


(lemma not-integerp-implies-not-equal-iplus (rewrite)
       (implies (not (integerp a))
                (equal (equal a (iplus b c))
                       f))
       ((use (integerp-iplus (x b) (y c)))
        (enable integerp)
        (do-not-induct t)))


(prove-lemma iplus-list-eval$-fringe (rewrite)
       ;; similar to IPLUS-TREE-IPLUS-FRINGE
       (equal (iplus-list (eval$ 'list (iplus-fringe x) a))
              (fix-int (eval$ t x a)))
       ((induct (iplus-fringe x))))

;; The following two lemmas aren't needed but they sure do
;; shorten the total proof time!!!


(prove-lemma iplus-ineg5-lemma-1 (rewrite)
  (implies (integerp x)
           (equal (equal x (iplus y (iplus (ineg z) w)))
                  (equal x (iplus (ineg z) (iplus y w))))))


(prove-lemma iplus-ineg5-lemma-2 (rewrite)
  (implies (and (integerp x) (integerp v))
           (equal (equal x (iplus (ineg z) v))
                  (equal (iplus x z) v))))


(lemma iplus-ineg5 (rewrite)
  (implies (integerp x)
           (equal (equal x (iplus y (iplus (ineg z) w)))
                  (equal (iplus x z) (iplus y w))))
  ((enable iplus-ineg5-lemma-1 iplus-ineg5-lemma-2 integerp-iplus)))
```

```
(disable iplus-ineg5-lemma-1)


(disable iplus-ineg5-lemma-2)


(lemma iplus-ineg6 (rewrite)
  (implies (integerp x)
           (equal (equal x (iplus y (iplus w (ineg z))))
                  (equal (iplus x z) (iplus y w))))
  ((use (iplus-ineg5)
        (commutativity-of-iplus (x w) (y (ineg z))))))


(prove-lemma eval$-iplus (rewrite)
  (implies (and (listp x)
                (equal (car x) 'iplus))
           (equal (eval$ t x a)
                  (iplus (eval$ t (cadr x) a)
                         (eval$ t (caddr x) a)))))


(prove-lemma iplus-ineg7 (rewrite)
  (equal (equal 0 (iplus x (ineg y)))
         (equal (fix-int y) (fix-int x)))
  ((enable-theory integer-defns)))


(prove-lemma correctness-of-cancel-iplus ((meta equal))
       (equal (eval$ t x a)
              (eval$ t (cancel-iplus x) a)))


(disable iplus-ineg5)


(disable iplus-ineg6)

; ----------------------------------------------------------------------------
; Cancel IPLUS from ILESSP
; ----------------------------------------------------------------------------

;; This is similar to the cancellation of IPLUS terms from equalities,
;; handled above, and uses many of the same lemmas.  A small but definite
;; difference however is that for ILESSP we don't have to fix integers.

;; By luck we have that iplus-tree-rec is appropriate here, since
;; the lemma eval$-iplus-tree-rec shows that it (accidentally) behaves
;; properly on the empty list.
```

```
(prove-lemma ilessp-fix-int-1 (rewrite)
  (equal (ilessp (fix-int x) y)
         (ilessp x y))
  ((enable-theory integer-defns)))


(prove-lemma ilessp-fix-int-2 (rewrite)
  (equal (ilessp x (fix-int y))
         (ilessp x y))
  ((enable-theory integer-defns)))

;; Perhaps the easiest approach is to do everything with respect to the
;; same IPLUS-TREE function that we used before, and then once the
;; supposed meta-lemma is proved, go back and show that we get the
;; same answer if we use a version that doesn't fix-int singleton fringes.


(defn make-cancel-iplus-inequality-1 (x y)
  ;; x and y are term lists
  (list (quote ilessp)
        (iplus-tree (bagdiff
                     x
                     (bagint x y)))
        (iplus-tree (bagdiff
                     y
                     (bagint x y)))))

#| The function below has no AND or OR, for efficiency

(defn cancel-iplus-ilessp-1 (x)
  (if (and (listp x)
           (equal (car x) (quote ilessp)))
      (make-cancel-iplus-inequality-1 (iplus-fringe (cadr x))
                                      (iplus-fringe (caddr x)))
    x))
|#


(DEFN CANCEL-IPLUS-ILESSP-1 (X)
  (IF (LISTP X)
      (IF (EQUAL (CAR X) 'ILESSP)
          (MAKE-CANCEL-IPLUS-INEQUALITY-1 (IPLUS-FRINGE (CADR X))
                                          (IPLUS-FRINGE (CADDR X)))
          X)
      X))

;; Notice that IPLUS-TREE-NO-FIX-INT is currently enabled, which is
;; good since we want to use EVAL$-IPLUS-TREE-NO-FIX-INT for now.


(prove-lemma lessp-difference-plus-arg1 (rewrite)
  (equal (lessp w (difference (plus w y) x))
         (lessp x y)))
```

```
(prove-lemma lessp-difference-plus-arg1-commuted (rewrite)
  (equal (lessp w (difference (plus y w) x))
         (lessp x y)))


(prove-lemma iplus-cancellation-1-for-ilessp (rewrite)
       (equal (ilessp (iplus a b) (iplus a c))
              (ilessp b c))
       ((enable-theory integer-defns)))


(prove-lemma iplus-cancellation-2-for-ilessp (rewrite)
       (equal (ilessp (iplus b a) (iplus c a))
              (ilessp b c)))


(prove-lemma correctness-of-cancel-iplus-ilessp-lemma nil
       (equal (eval$ t x a)
              (eval$ t (cancel-iplus-ilessp-1 x) a)))


(defn iplus-tree-no-fix-int (l)
  (if (listp l)
      (iplus-tree-rec l)
    (quote (quote 0))))


(prove-lemma eval$-ilessp-iplus-tree-no-fix-int (rewrite)
  (equal (ilessp (eval$ t (iplus-tree-no-fix-int x) a)
                 (eval$ t (iplus-tree-no-fix-int y) a))
         (ilessp (eval$ t (iplus-tree x) a)
                 (eval$ t (iplus-tree y) a))))


(disable iplus-tree-no-fix-int)


(lemma make-cancel-iplus-inequality-simplifier (rewrite)
       (equal (eval$ t (make-cancel-iplus-inequality-1 x y) a)
              (eval$ t (list (quote ilessp)
                             (iplus-tree-no-fix-int (bagdiff
                                                     x
                                                     (bagint x y)))
                             (iplus-tree-no-fix-int (bagdiff
                                                     y
                                                     (bagint x y)))) a))
       ((enable make-cancel-iplus-inequality-1
                eval$-ilessp-iplus-tree-no-fix-int)
        (disable eval$)))

#| The function below has no AND or OR, for efficiency
```

```
(defn cancel-iplus-ilessp (x)
  (if (and (listp x)
           (equal (car x) (quote ilessp)))
      (let ((x1 (iplus-fringe (cadr x)))
            (y1 (iplus-fringe (caddr x))))
        (let ((bagint (bagint x1 y1)))
          (if (listp bagint)
              ;; I check (listp bagint) only for efficiency
              (list (quote ilessp)
                    (iplus-tree-no-fix-int (bagdiff x1 bagint))
                    (iplus-tree-no-fix-int (bagdiff y1 bagint)))
            x)))
    x))
|#

;; **** Should perhaps check that some argument of the ILESSP has function
;; symbol IPLUS, or else we may wind up dealing with (ILESSP 0 0).  That should
;; be harmless enough, though, even if *1*IPLUS is disabled; we'll just get the
;; same term back, the hard way.


(DEFN CANCEL-IPLUS-ILESSP (X)
  (IF (LISTP X)
      (IF (EQUAL (CAR X) 'ILESSP)
          (IF (LISTP (BAGINT (IPLUS-FRINGE (CADR X))
                             (IPLUS-FRINGE (CADDR X))))
              (LIST 'ILESSP
                    (IPLUS-TREE-NO-FIX-INT
                     (BAGDIFF (IPLUS-FRINGE (CADR X))
                              (BAGINT (IPLUS-FRINGE (CADR X))
                                      (IPLUS-FRINGE (CADDR X)))))
                    (IPLUS-TREE-NO-FIX-INT
                     (BAGDIFF (IPLUS-FRINGE (CADDR X))
                              (BAGINT (IPLUS-FRINGE (CADR X))
                                      (IPLUS-FRINGE (CADDR X))))))
              X)
          X)
      X))


(disable make-cancel-iplus-inequality-1)


(prove-lemma correctness-of-cancel-iplus-ilessp ((meta ilessp))
       (equal (eval$ t x a)
              (eval$ t (cancel-iplus-ilessp x) a))
       ((use (correctness-of-cancel-iplus-ilessp-lemma))))

; ---------- Multiplication ----------
```

```
(lemma itimes-zero1 (rewrite)
       (implies (equal (fix-int x) 0)
                (equal (itimes x y) 0))
       ((enable itimes times fix-int integerp)
        (do-not-induct t)))


(prove-lemma itimes-0-left (rewrite)
       (equal (itimes 0 y) 0))

;; I don't want the backchaining to slow down the prover.

(disable itimes-zero1)


(lemma itimes-zero2 (rewrite)
       (implies (equal (fix-int y) 0)
                (equal (itimes x y) 0))
       ((enable itimes fix-int integerp times-zero)
        (do-not-induct t)))


(prove-lemma itimes-0-right (rewrite)
       (equal (itimes x 0) 0))

;; I don't want the backchaining to slow down the prover.

(disable itimes-zero2)


(lemma itimes-fix-int1 (rewrite)
       (equal (itimes (fix-int a) b)
              (itimes a b))
       ((enable itimes fix-int integerp)
        (do-not-induct t)))


(lemma itimes-fix-int2 (rewrite)
       (equal (itimes a (fix-int b))
              (itimes a b))
       ((enable itimes fix-int integerp times-zero)
        (do-not-induct t)))


(lemma commutativity-of-itimes (rewrite)
       (equal (itimes x y) (itimes y x))
       ((enable itimes fix-int integerp)
        (enable-theory multiplication)
        (do-not-induct t)))
```

```
(lemma itimes-distributes-over-iplus-proof ()
       (equal (itimes x (iplus y z))
              (iplus (itimes x y) (itimes x z)))
       ((enable itimes iplus integerp fix-int
                commutativity2-of-iplus associativity-of-iplus)
        (enable-theory multiplication addition)
        (do-not-induct t)))


(lemma itimes-distributes-over-iplus (rewrite)
       (and (equal (itimes x (iplus y z))
                   (iplus (itimes x y) (itimes x z)))
            (equal (itimes (iplus x y) z)
                   (iplus (itimes x z) (itimes y z))))
       ((use (itimes-distributes-over-iplus-proof (x x) (y y) (z z))
             (itimes-distributes-over-iplus-proof (x z) (y x) (z y)))
        (enable commutativity-of-itimes)))


(lemma commutativity2-of-itimes (rewrite)
       (equal (itimes x (itimes y z))
              (itimes y (itimes x z)))
       ((enable itimes integerp fix-int)
        (enable-theory multiplication)
        (do-not-induct t)))


(lemma associativity-of-itimes (rewrite)
       (equal (itimes (itimes x y) z)
              (itimes x (itimes y z)))
       ((enable itimes integerp fix-int)
        (enable-theory multiplication)
        (do-not-induct t)))


(lemma equal-itimes-0 (rewrite)
       (equal (equal (itimes x y) 0)
              (or (equal (fix-int x) 0)
                  (equal (fix-int y) 0)))
       ((enable itimes integerp fix-int)
        (enable-theory multiplication)
        (do-not-induct t)))


(lemma equal-itimes-1 (rewrite)
       (equal (equal (itimes a b) 1)
              (or (and (equal a 1)
                       (equal b 1))
                  (and (equal a -1)
                       (equal b -1))))
       ((enable itimes integerp fix-int)
        (enable-theory multiplication)
        (do-not-induct t)))
```

```
(lemma equal-itimes-minus-1 (rewrite)
       (equal (equal (itimes a b) -1)
              (or (and (equal a -1)
                       (equal b 1))
                  (and (equal a 1)
                       (equal b -1))))
       ((enable itimes integerp fix-int)
        (enable-theory multiplication)
        (do-not-induct t)))


(lemma itimes-1-arg1 (rewrite)
       (equal (itimes 1 x) (fix-int x))
       ((enable integerp fix-int itimes)
        (enable-theory multiplication)
        (do-not-induct t)))

; ---------- Division ----------


(lemma quotient-remainder-uniqueness ()
       (implies (and (equal a (plus r (times b q)))
                     (lessp r b))
                (and (equal (fix r) (remainder a b))
                     (equal (fix q) (quotient a b))))
       ((enable-theory naturals)
        (enable remainder quotient)))


; We want to define IQUOTIENT and IREMAINDER. The standard approach to
; integer division derives from from the following theorem.
;
; Division Theorem:
; For all integers i,j, j not 0, there exist unique integers q and r
; which satisfy  i = jq + r,  0 <= r < |j|.
;
; The functions IQUOTIENT and IREMAINDER are intended to compute q and r.
; Therefore, to be satisfied that we have the right definitions, we must
; prove the above theorem.


(prove-lemma division-theorem-part1 ()
       (implies (integerp i)
                (equal (iplus (iremainder i j) (itimes j (iquotient i j)))
                       i)))


(prove-lemma division-theorem-part2 ()
       (implies (and (integerp j)
                     (not (equal j 0)))
                (not (ilessp (iremainder i j) 0)))
       ((enable-theory integer-defns)))
```

```
(prove-lemma division-theorem-part3 ()
       (implies (and (integerp j)
                     (not (equal j 0)))
                (ilessp (iremainder i j) (iabs j)))
       ((enable-theory integer-defns)))


(lemma division-theorem ()
       (implies (and (integerp i)
                     (integerp j)
                     (not (equal j 0)))
                (and (equal (iplus (iremainder i j) (itimes j (iquotient i j)))
                            i)
                     (not (ilessp (iremainder i j) 0))
                     (ilessp (iremainder i j) (iabs j))))
       ((use (division-theorem-part1 (i i) (j j))
             (division-theorem-part2 (i i) (j j))
             (division-theorem-part3 (i i) (j j)))))


(lemma quotient-difference-lessp-arg2 (rewrite)
       (implies (and (equal (remainder a c) 0)
                     (lessp b c))
                (equal (quotient (difference a b) c)
                       (if (zerop b)
                           (quotient a c)
                           (if (lessp b a)
                               (difference (quotient a c)
                                           (add1 (quotient b c)))
                               0))))
       ((enable-theory naturals)
        (do-not-induct t)))
```

```
(lemma iquotient-iremainder-uniqueness ()
       (implies (and (integerp i)
                     (integerp j)
                     (integerp r)
                     (integerp q)
                     (not (equal j 0))
                     (equal i (iplus r (itimes j q)))
                     (not (ilessp r 0))
                     (ilessp r (iabs j)))
                (and (equal r (iremainder i j))
                     (equal q (iquotient i j))))
       ((enable iremainder iabs idifference iplus ineg
                fix-int itimes iquotient ilessp integerp
                quotient-difference-lessp-arg2)
        (enable-theory naturals)
        (do-not-induct t)))

; It turns out that in computer arithmetic, notions of division other than that
; given by the division theorem are used. Two in particular, called
; "truncate towards negative infinity" and "truncate towards zero" are common.
; We present their definitions here.

; Division Theorem (truncate towards negative infinity variant):
;
; For all integers i,j, j not 0, there exist unique integers q and r
; which satisfy
;                i = jq + r,  0 <= r <  j  (j > 0)
;                             j <  r <= 0  (j < 0)
;
; In this version the integer quotient of two integers is the integer floor
; of the real quotient of the integers. The remainder has the sign of the
; divisor. The functions IDIV and IMOD are intended to compute q and r.
; Therefore, to be satisfied that we have the right definitions, we must
; prove the above theorem.


(prove-lemma division-theorem-for-truncate-to-neginf-part1 ()
       (implies (integerp i)
                (equal (iplus (imod i j) (itimes j (idiv i j)))
                       i))
       ((enable-theory integer-defns)))


(lemma division-theorem-for-truncate-to-neginf-part2 ()
       (implies (ilessp 0 j)
                (and (not (ilessp (imod i j) 0))
                     (ilessp (imod i j) j)))
       ((enable imod ilessp idifference iplus ineg
                itimes idiv integerp fix-int)
        (enable-theory naturals)
        (do-not-induct t)))
```

```
(lemma division-theorem-for-truncate-to-neginf-part3 ()
       (implies (and (integerp j)
                     (ilessp j 0))
                (and (not (ilessp 0 (imod i j)))
                     (ilessp j (imod i j))))
       ((enable imod ilessp idifference iplus ineg
                itimes idiv integerp fix-int)
        (enable-theory naturals)
        (do-not-induct t)))


(lemma division-theorem-for-truncate-to-neginf ()
       (implies (and (integerp i)
                     (integerp j)
                     (not (equal j 0)))
                (and (equal (iplus (imod i j) (itimes j (idiv i j)))
                            i)
                     (if (ilessp 0 j)
                         (and (not (ilessp (imod i j) 0))
                              (ilessp (imod i j) j))
                         (and (not (ilessp 0 (imod i j)))
                              (ilessp j (imod i j))))))
       ((use (division-theorem-for-truncate-to-neginf-part1 (i i) (j j))
             (division-theorem-for-truncate-to-neginf-part2 (i i) (j j))
             (division-theorem-for-truncate-to-neginf-part3 (i i) (j j)))
        (enable integerp ilessp)
        (do-not-induct t)))
```

```
(lemma idiv-imod-uniqueness ()
       (implies (and (integerp i)
                     (integerp j)
                     (integerp r)
                     (integerp q)
                     (not (equal j 0))
                     (equal i (iplus r (itimes j q)))
                     (if (ilessp 0 j)
                         (and (not (ilessp r 0))
                              (ilessp r j))
                         (and (not (ilessp 0 r))
                              (ilessp j r))))
                (and (equal r (imod i j))
                     (equal q (idiv i j))))
       ((enable imod iabs idifference iplus ineg
                fix-int itimes idiv ilessp integerp
                ;lessp-plus-times-crock
                ;lessp-times-crock1
                ;lessp-times-crock2
                ;lessp-times-crock3
                ;lessp-times-crock4
                quotient-difference-lessp-arg2)
        (enable-theory naturals)
        (do-not-induct t)))

; Division Theorem (truncate towards zero variant):
;
; For all integers i,j, j not 0, there exist unique integers q and r
; which satisfy
;               i = jq + r,     0 <=  r  <  |j|  (i => 0)
;                            -|j| <   r  <= 0    (i < 0)
;
; In this version (iquo, irem), the integer quotient of two integers
; is the integer floor of the real quotient of the integers, if the
; real quotient is positive. If the real quotient is negative, the
; integer quotient is the integer ceiling of the real quotient. The
; remainder has the sign of the dividend. The functions IQUO and IREM
; are intended to compute q and r. Therefore, to be satisfied that we
; have the right definitions, we must prove the above theorem.


(prove-lemma division-theorem-for-truncate-to-zero-part1 ()
       (implies (integerp i)
                (equal (iplus (irem i j) (itimes j (iquo i j)))
                       i))
       ((enable-theory integer-defns)))
```

```
(prove-lemma division-theorem-for-truncate-to-zero-part2 ()
       (implies (and (integerp i)
                     (integerp j)
                     (not (equal j 0))
                     (not (ilessp i 0)))
                (and (not (ilessp (irem i j) 0))
                     (ilessp (irem i j) (iabs j))))
       ((enable-theory integer-defns)))


(prove-lemma division-theorem-for-truncate-to-zero-part3 ()
       (implies (and (integerp i)
                     (integerp j)
                     (not (equal j 0))
                     (ilessp i 0))
                (and (not (ilessp 0 (irem i j)))
                     (ilessp (ineg (iabs j)) (irem i j))))
       ((enable-theory integer-defns)))


(lemma division-theorem-for-truncate-to-zero ()
       (implies (and (integerp i)
                     (integerp j)
                     (not (equal j 0)))
                (and (equal (iplus (irem i j) (itimes j (iquo i j)))
                            i)
                     (if (not (ilessp i 0))
                         (and (not (ilessp (irem i j) 0))
                              (ilessp (irem i j) (iabs j)))
                         (and (not (ilessp 0 (irem i j)))
                              (ilessp (ineg (iabs j)) (irem i j))))))
       ((use (division-theorem-for-truncate-to-zero-part1 (i i) (j j))
             (division-theorem-for-truncate-to-zero-part2 (i i) (j j))
             (division-theorem-for-truncate-to-zero-part3 (i i) (j j)))
        (enable integerp ilessp)
        (do-not-induct t)))
```

```
(prove-lemma iquo-irem-uniqueness ()
       (implies (and (integerp i)
                     (integerp j)
                     (integerp r)
                     (integerp q)
                     (not (equal j 0))
                     (equal i (iplus r (itimes j q)))
                     (if (not (ilessp i 0))
                         (and (not (ilessp r 0))
                              (ilessp r (iabs j)))
                         (and (not (ilessp 0 r))
                              (ilessp (ineg (iabs j)) r))))
                (and (equal r (irem i j))
                     (equal q (iquo i j))))
       ((enable-theory integer-defns)))

; ---------- Multiplication Facts


(prove-lemma itimes-ineg-1 (rewrite)
  (equal (itimes (ineg x) y)
         (ineg (itimes x y)))
  ((enable-theory integer-defns)))


(prove-lemma itimes-ineg-2 (rewrite)
  (equal (itimes x (ineg y))
         (ineg (itimes x y)))
  ((enable-theory integer-defns)))


(prove-lemma itimes-cancellation-1 (rewrite)
       (equal (equal (itimes a b) (itimes a c))
              (or (equal (fix-int a) 0)
                  (equal (fix-int b) (fix-int c))))
       ((enable-theory integer-defns)))


(lemma itimes-cancellation-2 (rewrite)
       (equal (equal (itimes b a) (itimes c a))
              (or (equal (fix-int a) 0)
                  (equal (fix-int b) (fix-int c))))
       ((use (itimes-cancellation-1))
        (enable commutativity-of-itimes)))
```

```
(lemma itimes-cancellation-3 (rewrite)
       (equal (equal (itimes a b) (itimes c a))
              (or (equal (fix-int a) 0)
                  (equal (fix-int b) (fix-int c))))
       ((use (itimes-cancellation-1))
        (enable commutativity-of-itimes)))

; ---------- Division Facts


(lemma integerp-iquotient (rewrite)
       (integerp (iquotient i j))
       ((enable integerp iquotient fix-int)
        (do-not-induct t)))


(lemma integerp-iremainder (rewrite)
       (integerp (iremainder i j))
       ((enable iremainder integerp-idifference)
        (do-not-induct t)))


(lemma integerp-idiv (rewrite)
       (integerp (idiv i j))
       ((enable integerp idiv fix-int)
        (do-not-induct t)))


(lemma integerp-imod (rewrite)
       (integerp (imod i j))
       ((enable imod integerp-idifference)
        (do-not-induct t)))


(lemma integerp-iquo (rewrite)
       (integerp (iquo i j))
       ((enable integerp iquo fix-int)
        (do-not-induct t)))


(lemma integerp-irem (rewrite)
       (integerp (irem i j))
       ((enable irem integerp-idifference)
        (do-not-induct t)))


(lemma iquotient-fix-int1 (rewrite)
       (equal (iquotient (fix-int i) j)
              (iquotient i j))
       ((enable integerp iquotient fix-int)
        (do-not-induct t)))
```

```
(lemma iquotient-fix-int2 (rewrite)
       (equal (iquotient i (fix-int j))
              (iquotient i j))
       ((enable integerp iquotient fix-int)
        (do-not-induct t)))


(lemma iremainder-fix-int1 (rewrite)
       (equal (iremainder (fix-int i) j)
              (iremainder i j))
       ((enable iremainder idifference-fix-int1 iquotient-fix-int1)
        (do-not-induct t)))


(lemma iremainder-fix-int2 (rewrite)
       (equal (iremainder i (fix-int j))
              (iremainder i j))
       ((enable iremainder itimes-fix-int1 iquotient-fix-int2)
        (do-not-induct t)))


(lemma idiv-fix-int1 (rewrite)
       (equal (idiv (fix-int i) j)
              (idiv i j))
       ((enable integerp idiv fix-int)
        (do-not-induct t)))


(lemma idiv-fix-int2 (rewrite)
       (equal (idiv i (fix-int j))
              (idiv i j))
       ((enable integerp idiv fix-int)
        (do-not-induct t)))


(lemma imod-fix-int1 (rewrite)
       (equal (imod (fix-int i) j)
              (imod i j))
       ((enable imod fix-int-fix-int idiv-fix-int1)
        (do-not-induct t)))


(lemma imod-fix-int2 (rewrite)
       (equal (imod i (fix-int j))
              (imod i j))
       ((enable imod itimes-fix-int1 idiv-fix-int2)
        (do-not-induct t)))
```

```
(lemma iquo-fix-int1 (rewrite)
       (equal (iquo (fix-int i) j)
              (iquo i j))
       ((enable integerp iquo fix-int)
        (do-not-induct t)))


(lemma iquo-fix-int2 (rewrite)
       (equal (iquo i (fix-int j))
              (iquo i j))
       ((enable integerp iquo fix-int)
        (do-not-induct t)))


(lemma irem-fix-int1 (rewrite)
       (equal (irem (fix-int i) j)
              (irem i j))
       ((enable irem fix-int-fix-int iquo-fix-int1)
        (do-not-induct t)))


(lemma irem-fix-int2 (rewrite)
       (equal (irem i (fix-int j))
              (irem i j))
       ((enable irem itimes-fix-int1 iquo-fix-int2)
        (do-not-induct t)))


(lemma fix-int-iquotient (rewrite)
       (equal (fix-int (iquotient i j))
              (iquotient i j))
       ((enable integerp iquotient fix-int)
        (do-not-induct t)))


(lemma fix-int-iremainder (rewrite)
       (equal (fix-int (iremainder i j))
              (iremainder i j))
       ((enable iremainder fix-int-idifference)
        (do-not-induct t)))


(lemma fix-int-idiv (rewrite)
       (equal (fix-int (idiv i j))
              (idiv i j))
       ((enable integerp idiv fix-int)
        (do-not-induct t)))
```

```
(lemma fix-int-imod (rewrite)
       (equal (fix-int (imod i j))
              (imod i j))
       ((enable imod fix-int-idifference)
        (do-not-induct t)))


(lemma fix-int-iquo (rewrite)
       (equal (fix-int (iquo i j))
              (iquo i j))
       ((enable integerp iquo fix-int)
        (do-not-induct t)))


(lemma fix-int-irem (rewrite)
       (equal (fix-int (irem i j))
              (irem i j))
       ((enable irem fix-int-idifference)
        (do-not-induct t)))


(disable iquotient)

(disable iremainder)

(disable idiv)

(disable imod)

(disable iquo)

(disable irem)

; ---------- Meta lemma for itimes cancellation

;; I tried to adapt this somewhat from corresponding meta lemmas in
;; naturals library, but it seemed to get hairy.  So instead I'll try
;; to parallel the development I gave for IPLUS.  I'll be lazier here
;; about efficiency, so I'll use a completely analogous definition of
;; itimes-tree.  Notice that I've avoided the IZEROP-TREE approach
;; from the naturals version, in that I simply create the appropriate
;; common fringe into a product and say that this product is non-zero
;; when dividing both sides by it.  It can then be up to the user
;; whether or not to enable the (meta or rewrite) rule that says that
;; izerop of a product reduces to the disjunction of izerop of the
;; factors.

#| The function below has no AND or OR, for efficiency
```

```
(defn itimes-fringe (x)
  (if (and (listp x)
           (equal (car x)
                  (quote itimes)))
      (append (itimes-fringe (cadr x))
              (itimes-fringe (caddr x)))
      (cons x nil)))
|#


(DEFN ITIMES-FRINGE (X)
  (IF (LISTP X)
      (IF (EQUAL (CAR X) 'ITIMES)
          (APPEND (ITIMES-FRINGE (CADR X))
                  (ITIMES-FRINGE (CADDR X)))
          (LIST X))
      (LIST X)))


(defn itimes-tree-rec (l)
  (if (nlistp (cdr l))
      (car l)
    (list (quote itimes)
          (car l)
          (itimes-tree-rec (cdr l)))))


(defn itimes-tree (l)
  (if (listp l)
      (if (listp (cdr l))
          (itimes-tree-rec l)
        (list (quote fix-int)
              (car l)))
    (quote (quote 1))))


(defn itimes-list (x)
  (if (listp x)
      (itimes (car x)
              (itimes-list (cdr x)))
    1))


(prove-lemma integerp-itimes-list (rewrite)
  (integerp (itimes-list x)))
```

```
(prove-lemma eval$-itimes-tree-rec (rewrite)
  (implies (listp x)
           (equal (eval$ t (itimes-tree-rec x) a)
                  (if (listp (cdr x))
                      (itimes-list (eval$ 'list x a))
                    (eval$ t (car x) a)))))

;; The following allows us to pretty much ignore itimes-tree forever.  (Notice
;; that it is disabled immediately below.)


(prove-lemma eval$-itimes-tree (rewrite)
  (equal (eval$ t (itimes-tree x) a)
         (itimes-list (eval$ 'list x a))))


(disable itimes-tree) ;; because we want to use EVAL$-ITIMES-TREE for now


(defn make-cancel-itimes-equality (x y in-both)
  ;; x and y are term lists and for efficiency we pass in-both as their bagint,
  ;; which is a listp.
  (list 'if
        (list 'equal (itimes-tree in-both) ''0)
        (list 'quote t)
        (list (quote equal)
              (itimes-tree (bagdiff x in-both))
              (itimes-tree (bagdiff y in-both)))))

#| The function below has no AND or OR, for efficiency
```

```
(defn cancel-itimes (x)
  (if (and (listp x)
           (equal (car x) (quote equal)))
      (if (and (listp (cadr x))
               (equal (caadr x) (quote itimes))
               (listp (caddr x))
               (equal (caaddr x) (quote itimes)))
          (if (listp (bagint (itimes-fringe (cadr x))
                             (itimes-fringe (caddr x))))
              (make-cancel-itimes-equality (itimes-fringe (cadr x))
                                           (itimes-fringe (caddr x))
                                           (bagint (itimes-fringe (cadr x))
                                                   (itimes-fringe (caddr x))))
            x)
        (if (and (listp (cadr x))
                 (equal (caadr x) (quote itimes)))
            ;; We don't want to introduce the IF below unless something
            ;; is "gained", or else we may get into an infinite rewriting loop.
            (if (member (caddr x) (itimes-fringe (cadr x)))
                (list (quote if)
                      (list (quote integerp) (caddr x))
                      (make-cancel-itimes-equality (itimes-fringe (cadr x))
                                                   (list (caddr x))
                                                   (list (caddr x)))
                      (list (quote quote) f))
              x)
          (if (and (listp (caddr x))
                   (equal (caaddr x) (quote itimes)))
              (if (member (cadr x) (itimes-fringe (caddr x)))
                  (list (quote if)
                        (list (quote integerp) (cadr x))
                        (make-cancel-itimes-equality (list (cadr x))
                                                     (itimes-fringe (caddr x))
                                                     (list (cadr x)))
                        (list (quote quote) f))
                x)
            x)))
    x))
|#
```

```
(DEFN CANCEL-ITIMES (X)
  (IF (LISTP X)
      (IF (EQUAL (CAR X) 'EQUAL)
          (COND
           ((LISTP (CADR X))
            (COND
             ((EQUAL (CAADR X) 'ITIMES)
              (COND
               ((LISTP (CADDR X))
                (COND
                 ((EQUAL (CAADDR X) 'ITIMES)
                  (IF (LISTP (BAGINT (ITIMES-FRINGE (CADR X))
                                     (ITIMES-FRINGE (CADDR X))))
                      (MAKE-CANCEL-ITIMES-EQUALITY
                       (ITIMES-FRINGE (CADR X))
                       (ITIMES-FRINGE (CADDR X))
                       (BAGINT (ITIMES-FRINGE (CADR X))
                               (ITIMES-FRINGE (CADDR X))))
                      X))
                 ((MEMBER (CADDR X) (ITIMES-FRINGE (CADR X)))
                  (LIST 'IF (LIST 'INTEGERP (CADDR X))
                        (MAKE-CANCEL-ITIMES-EQUALITY
                         (ITIMES-FRINGE (CADR X))
                         (LIST (CADDR X)) (LIST (CADDR X)))
                        (LIST 'QUOTE F)))
                 (T X)))
               ((MEMBER (CADDR X) (ITIMES-FRINGE (CADR X)))
                (LIST 'IF (LIST 'INTEGERP (CADDR X))
                      (MAKE-CANCEL-ITIMES-EQUALITY
                       (ITIMES-FRINGE (CADR X))
                       (LIST (CADDR X)) (LIST (CADDR X)))
                      (LIST 'QUOTE F)))
               (T X)))
             ((LISTP (CADDR X))
              (IF (EQUAL (CAADDR X) 'ITIMES)
                  (IF (MEMBER (CADR X) (ITIMES-FRINGE (CADDR X)))
                      (LIST 'IF (LIST 'INTEGERP (CADR X))
                            (MAKE-CANCEL-ITIMES-EQUALITY
                             (LIST (CADR X))
                             (ITIMES-FRINGE (CADDR X))
                             (LIST (CADR X)))
                            (LIST 'QUOTE F))
                      X)
                  X))
             (T X)))
           ((LISTP (CADDR X))
            (IF (EQUAL (CAADDR X) 'ITIMES)
                (IF (MEMBER (CADR X) (ITIMES-FRINGE (CADDR X)))
                    (LIST 'IF (LIST 'INTEGERP (CADR X))
                          (MAKE-CANCEL-ITIMES-EQUALITY
                           (LIST (CADR X))
                           (ITIMES-FRINGE (CADDR X))
                           (LIST (CADR X)))
                          (LIST 'QUOTE F))
```

```
                    X)
                X))
           (T X))
          X)
      X))


(prove-lemma itimes-list-append (rewrite)
  (equal (itimes-list (append x y))
         (itimes (itimes-list x) (itimes-list y))))


(prove-lemma itimes-list-eval$-fringe (rewrite)
       ;; similar to ITIMES-TREE-ITIMES-FRINGE
       (equal (itimes-list (eval$ 'list (itimes-fringe x) a))
              (fix-int (eval$ t x a)))
       ((induct (itimes-fringe x))))


(prove-lemma integerp-eval$-itimes (rewrite)
  (implies (equal (car x) 'itimes)
           (integerp (eval$ t x a))))


(lemma not-integerp-implies-not-equal-itimes (rewrite)
       (implies (not (integerp a))
                (equal (equal a (itimes b c))
                       f))
       ((use (integerp-itimes (x b) (y c)))
        (enable integerp)
        (do-not-induct t)))


(prove-lemma itimes-list-eval$-delete (rewrite)
  (implies (member z y)
           (equal (itimes-list (eval$ 'list y a))
                  (itimes (eval$ t z a)
                          (itimes-list (eval$ 'list (delete z y) a))))))


(prove-lemma itimes-list-bagdiff (rewrite)
       (implies (subbagp x y)
                (equal (itimes-list (eval$ 'list y a))
                       (itimes (itimes-list (eval$ 'list (bagdiff y x) a))
                               (itimes-list (eval$ 'list x a)))))
       ((induct (bagdiff y x))))
```

```
(prove-lemma equal-itimes-list-eval$-list-delete (rewrite)
  (implies (and (member c y)
                (not (equal (fix-int (eval$ t c a)) 0)))
           (equal (equal x (itimes-list (eval$ 'list (delete c y) a)))
                  (and (integerp x)
                       (equal (itimes x (eval$ t c a))
                              (itimes-list (eval$ 'list y a)))))))


(disable itimes-list-eval$-delete)

;; I had trouble with the clausifier (thanks, J, for pointing that out
;; as the source of my trouble) in the proof of the meta lemma -- it's
;; getting rid of a case split.  So, I'll proceed by reducing
;; cancel-itimes in each case; see lemma eval$-make-cancel-itimes-equality
;; (and its -1 and -2 versions).


(prove-lemma member-append (rewrite)
  (equal (member a (append x y))
         (or (member a x) (member a y))))


(prove-lemma member-izerop-itimes-fringe (rewrite)
  (implies (and (member z (itimes-fringe x))
                (equal (fix-int (eval$ t z a)) 0))
           (equal (fix-int (eval$ t x a)) 0))
  ((induct (itimes-fringe x))))


(prove-lemma correctness-of-cancel-itimes-hack-1 (rewrite)
  (implies (and (member w
                        (itimes-fringe (cons 'itimes x1)))
                (equal (fix-int (eval$ t w a)) 0)
                (not (equal (fix-int (eval$ t (car x1) a)) 0)))
           (equal (fix-int (eval$ t (cadr x1) a)) 0)))


(enable eval$-equal)


(prove-lemma eval$-make-cancel-itimes-equality (rewrite)
  (equal (eval$ t (make-cancel-itimes-equality x y in-both) a)
         (if (eval$ t (list 'equal (itimes-tree in-both) ''0) a)
             t
           (equal
              (itimes-list (eval$ 'list (bagdiff x in-both) a))
              (itimes-list (eval$ 'list (bagdiff y in-both) a))))))
```

```
(disable make-cancel-itimes-equality)

;; Here's a special case that I hope helps with the clausifier problem.
;; The lemma above seems necessary for its proof.


(prove-lemma eval$-make-cancel-itimes-equality-1 (rewrite)
  (equal (eval$ t (make-cancel-itimes-equality (list x) y (list x)) a)
         (if (equal (fix-int (eval$ t x a)) 0)
             t
           (equal
              1
              (itimes-list (eval$ 'list (delete x y) a))))))


(prove-lemma equal-fix-int (rewrite)
  (equal (equal (fix-int x) x)
         (integerp x))
  ((enable-theory integer-defns)))

;; Here's another special case that I hope helps with the clausifier problem.


(prove-lemma eval$-make-cancel-itimes-equality-2 (rewrite)
  (equal (eval$ t (make-cancel-itimes-equality x (list y) (list y)) a)
         (if (equal (fix-int (eval$ t y a)) 0)
             t
           (equal
              1
              (itimes-list (eval$ 'list (delete y x) a))))))


(prove-lemma eval$-equal-itimes-tree-itimes-fringe-0 (rewrite)
  (implies (and (eval$ t
                       (list 'equal
                             (itimes-tree (itimes-fringe x))
                             ''0)
                       a)
                (equal (car x) 'itimes))
           (equal (eval$ t x a) 0)))


(prove-lemma izerop-eval-of-member-implies-itimes-list-0 (rewrite)
  (implies (and (member z y)
                (equal (fix-int (eval$ t z a)) 0))
           (equal (itimes-list (eval$ 'list y a))
                  0)))

#| The function below has no AND or OR, for efficiency
```

```
(defn subsetp (x y)
  (if (nlistp x)
      t
    (and (member (car x) y)
         (subsetp (cdr x) y))))
|#


(DEFN SUBSETP (X Y)
  (COND
   ((NLISTP X) T)
   ((MEMBER (CAR X) Y) (SUBSETP (CDR X) Y))
   (T F)))


(prove-lemma subsetp-implies-itimes-list-eval$-equals-0 (rewrite)
  (implies (and (subsetp x y)
                (equal (itimes-list (eval$ 'list x a))
                       0))
           (equal (itimes-list (eval$ 'list y a))
                  0)))


(prove-lemma subbagp-subsetp (rewrite)
  (implies (subbagp x y)
           (subsetp x y)))


(prove-lemma equal-0-itimes-list-eval$-bagint-1 (rewrite)
  (implies (equal (itimes-list (eval$ 'list (bagint x y) a))
                  0)
           (equal (itimes-list (eval$ 'list x a))
                  0))
  ((use (subsetp-implies-itimes-list-eval$-equals-0 (x (bagint x y)) (y x)))
   (disable subsetp-implies-itimes-list-eval$-equals-0)))


(prove-lemma equal-0-itimes-list-eval$-bagint-2 (rewrite)
  (implies (equal (itimes-list (eval$ 'list (bagint x y) a))
                  0)
           (equal (itimes-list (eval$ 'list y a))
                  0))
  ((use (subsetp-implies-itimes-list-eval$-equals-0 (x (bagint x y)) (y y)))
   (disable subsetp-implies-itimes-list-eval$-equals-0)))
```

```
(prove-lemma correctness-of-cancel-itimes-hack-2 (rewrite)
 (implies
   (and (listp u)
        (equal (car u) 'itimes)
        (listp v)
        (equal (car v) 'itimes)
        (not (equal (eval$ t u a)
                    (eval$ t v a))))
   (not (equal (itimes-list (eval$ 'list
                                  (bagdiff (itimes-fringe u)
                                           (bagint (itimes-fringe u)
                                                   (itimes-fringe v)))
                                  a))
               (itimes-list (eval$ 'list
                                  (bagdiff (itimes-fringe v)
                                           (bagint (itimes-fringe u)
                                                   (itimes-fringe v)))
                                  a)))))
 ((use (itimes-list-bagdiff
        (y (itimes-fringe u))
        (x (bagint (itimes-fringe u)
                   (itimes-fringe v)))
        (a a))
       (itimes-list-bagdiff
        (y (itimes-fringe v))
        (x (bagint (itimes-fringe u)
                   (itimes-fringe v)))
        (a a)))))

(prove-lemma correctness-of-cancel-itimes-hack-3-lemma (rewrite)
  (implies (and (equal u (itimes a b))
                (not (equal (fix-int a) 0)))
           (equal (equal u (itimes a c))
                  (equal (fix-int b) (fix-int c)))))
```

```
(prove-lemma correctness-of-cancel-itimes-hack-3 (rewrite)
 (implies
   (and (listp u)
        (equal (car u) 'itimes)
        (listp v)
        (equal (car v) 'itimes)
        (equal (eval$ t u a)
               (eval$ t v a))
        (not (eval$ t
                    (list 'equal
                          (itimes-tree (bagint (itimes-fringe u)
                                               (itimes-fringe v)))
                          ''0)
                    a)))
   (equal (equal (itimes-list (eval$ 'list
                                     (bagdiff (itimes-fringe u)
                                              (bagint (itimes-fringe u)
                                                      (itimes-fringe v)))
                                     a))
                 (itimes-list (eval$ 'list
                                     (bagdiff (itimes-fringe v)
                                              (bagint (itimes-fringe u)
                                                      (itimes-fringe v)))
                                     a)))
          t))
 ((use (itimes-list-bagdiff
        (y (itimes-fringe u))
        (x (bagint (itimes-fringe u)
                   (itimes-fringe v)))
        (a a))
       (itimes-list-bagdiff
        (y (itimes-fringe v))
        (x (bagint (itimes-fringe u)
                   (itimes-fringe v)))
        (a a)))))


(disable correctness-of-cancel-itimes-hack-3-lemma)


(prove-lemma correctness-of-cancel-itimes ((meta equal))
       (equal (eval$ t x a)
              (eval$ t (cancel-itimes x) a))
       ((do-not-induct t)))

; ---------- Meta lemma for itimes cancellation on ilessp terms

;; I'll try to keep this similar to the approach for equalities above,
;; modified as in the iplus case (i.e. no fix-int is necessary).

;; EVAL$-EQUAL is currently enabled, but that's OK.
```

```
(defn itimes-tree-no-fix-int (l)
  (if (listp l)
      (itimes-tree-rec l)
    (quote (quote 1))))

;; The following allows us to pretty much ignore
;; itimes-tree-no-fix-int forever.  (Notice that it is disabled
;; immediately below.)


(prove-lemma eval$-itimes-tree-no-fix-int-1 (rewrite)
  (equal (ilessp (eval$ t (itimes-tree-no-fix-int x) a)
                 y)
         (ilessp (eval$ t (itimes-tree x) a)
                 y)))


(prove-lemma eval$-itimes-tree-no-fix-int-2 (rewrite)
  (equal (ilessp y
                 (eval$ t (itimes-tree-no-fix-int x) a))
         (ilessp y
                 (eval$ t (itimes-tree x) a))))


(disable itimes-tree-no-fix-int)

;; We want to use EVAL$-ITIMES-TREE, and ITIMES-TREE is still disabled
;; so we're in good shape.


(defn make-cancel-itimes-inequality (x y in-both)
  ;; x and y are term lists and for efficiency we pass in-both as their bagint,
  ;; which is a listp.
  (list 'if
        (list 'ilessp (itimes-tree-no-fix-int in-both) ''0)
        (list (quote ilessp)
              (itimes-tree-no-fix-int (bagdiff y in-both))
              (itimes-tree-no-fix-int (bagdiff x in-both)))
        (list 'if
              (list 'ilessp ''0 (itimes-tree-no-fix-int in-both))
              (list (quote ilessp)
                    (itimes-tree-no-fix-int (bagdiff x in-both))
                    (itimes-tree-no-fix-int (bagdiff y in-both)))
              '(false))))

#| The function below has no AND or OR, for efficiency
```

```
(defn cancel-itimes-ilessp (x)
  (if (and (listp x)
           (equal (car x) (quote ilessp))
           (listp (bagint (itimes-fringe (cadr x)) (itimes-fringe (caddr x)))))
      (make-cancel-itimes-inequality (itimes-fringe (cadr x))
                                     (itimes-fringe (caddr x))
                                     (bagint (itimes-fringe (cadr x))
                                             (itimes-fringe (caddr x))))
    x))
|#


(DEFN CANCEL-ITIMES-ILESSP (X)
  (IF (LISTP X)
      (IF (EQUAL (CAR X) 'ILESSP)
          (IF (LISTP (BAGINT (ITIMES-FRINGE (CADR X))
                             (ITIMES-FRINGE (CADDR X))))
              (MAKE-CANCEL-ITIMES-INEQUALITY
               (ITIMES-FRINGE (CADR X))
               (ITIMES-FRINGE (CADDR X))
               (BAGINT (ITIMES-FRINGE (CADR X))
                       (ITIMES-FRINGE (CADDR X))))
              X)
          X)
      X))


(prove-lemma eval$-make-cancel-itimes-inequality (rewrite)
  (equal
   (eval$ t (make-cancel-itimes-inequality x y in-both) a)
   (if (eval$ t (list 'ilessp (itimes-tree-no-fix-int in-both) ''0) a)
       (ilessp (eval$ t (itimes-tree-no-fix-int (bagdiff y in-both)) a)
               (eval$ t (itimes-tree-no-fix-int (bagdiff x in-both)) a))
     (if (eval$ t (list 'ilessp ''0 (itimes-tree-no-fix-int in-both)) a)
         (ilessp (eval$ t (itimes-tree-no-fix-int (bagdiff x in-both)) a)
                 (eval$ t (itimes-tree-no-fix-int (bagdiff y in-both)) a))
       f))))


(disable make-cancel-itimes-inequality)


(prove-lemma listp-bagint-with-singleton-implies-member (rewrite)
  (implies (listp (bagint y (list z)))
           (member z y)))


(prove-lemma itimes-list-eval$-list-0 (rewrite)
  (implies (member 0 x)
           (equal (itimes-list (eval$ 'list x a))
                  0)))
```

```
(prove-lemma ilessp-itimes-right-positive nil
             (implies (ilessp 0 x)
                      (equal (ilessp y z)
                             (ilessp (itimes y x) (itimes z x))))
             ((enable-theory integer-defns)))


(prove-lemma correctness-of-cancel-itimes-ilessp-hack-1 (rewrite)
  (implies (and (subbagp bag x)
                (subbagp bag y)
                (ilessp 0 (itimes-list (eval$ 'list bag a))))
           (equal (ilessp (itimes-list (eval$ 'list
                                              (bagdiff x bag)
                                              a))
                          (itimes-list (eval$ 'list
                                              (bagdiff y bag)
                                              a)))
                  (ilessp (itimes-list (eval$ 'list x a))
                          (itimes-list (eval$ 'list y a)))))
  ((use (ilessp-itimes-right-positive
         (x (itimes-list (eval$ 'list bag a)))
         (y (itimes-list (eval$ 'list (bagdiff x bag) a)))
         (z (itimes-list (eval$ 'list (bagdiff y bag) a))))
        (itimes-list-bagdiff (y x) (x bag) (a a))
        (itimes-list-bagdiff (y y) (x bag) (a a)))))


(prove-lemma listp-bagint-with-singleton-member (rewrite)
  (equal (listp (bagint y (list z)))
         (member z y)))


(prove-lemma correctness-of-cancel-itimes-ilessp-hack-2-lemma (rewrite)
  (implies (member 0 (itimes-fringe w))
           (equal (eval$ t w a) 0))
  ((expand (itimes-fringe w))))


(prove-lemma correctness-of-cancel-itimes-ilessp-hack-2 (rewrite)
  (implies (member 0 (itimes-fringe w))
           (not (ilessp (eval$ t w a) 0))))


(disable correctness-of-cancel-itimes-ilessp-hack-2-lemma)

;;; Now hack-3 and hack-4 below are all that's left to prove before the
;;; main result.
```

```
(prove-lemma ilessp-trichotomy (rewrite)
  (implies (not (ilessp x y))
           (equal (ilessp y x)
                  (not (equal (fix-int x) (fix-int y)))))
  ((enable-theory integer-defns)))


(prove-lemma correctness-of-cancel-itimes-ilessp-hack-3-lemma-1 nil
  (implies (and (equal 0 (itimes-list (eval$ 'list bag a)))
                (subsetp bag z))
           (equal (itimes-list (eval$ 'list z a)) 0)))


(prove-lemma correctness-of-cancel-itimes-ilessp-hack-3-lemma-2 nil
  (implies (and (equal 0 (itimes-list (eval$ 'list bag a)))
                (subsetp bag (itimes-fringe x)))
           (equal (fix-int (eval$ t x a)) 0))
  ((use (correctness-of-cancel-itimes-ilessp-hack-3-lemma-1
         (z (itimes-fringe x))))))


(prove-lemma same-fix-int-implies-not-ilessp (rewrite)
  (implies (equal (fix-int x) (fix-int y))
           (not (ilessp x y)))
  ((enable-theory integer-defns)))


(prove-lemma correctness-of-cancel-itimes-ilessp-hack-3 (rewrite)
  (implies
   (and (not (ilessp (itimes-list (eval$ 'list
                                        bag
                                        a))
                     0))
        (not (ilessp 0
                     (itimes-list (eval$ 'list
                                        bag
                                        a))))
        (subbagp bag (itimes-fringe w))
        (subbagp bag (itimes-fringe v)))
   (not (ilessp (eval$ t w a)
                (eval$ t v a))))
  ((use (correctness-of-cancel-itimes-ilessp-hack-3-lemma-2
        (x w) (bag bag))
        (correctness-of-cancel-itimes-ilessp-hack-3-lemma-2
         (x v) (bag bag)))))


(prove-lemma ilessp-itimes-right-negative nil
             (implies (ilessp x 0)
                      (equal (ilessp y z)
                             (ilessp (itimes z x) (itimes y x))))
             ((enable-theory integer-defns)))
```

```
(prove-lemma correctness-of-cancel-itimes-ilessp-hack-4 (rewrite)
  (implies (and (subbagp bag x)
                (subbagp bag y)
                (ilessp (itimes-list (eval$ 'list bag a)) 0))
           (equal (ilessp (itimes-list (eval$ 'list
                                              (bagdiff x bag)
                                              a))
                          (itimes-list (eval$ 'list
                                              (bagdiff y bag)
                                              a)))
                  (ilessp (itimes-list (eval$ 'list y a))
                          (itimes-list (eval$ 'list x a)))))
  ((use (ilessp-itimes-right-negative
         (x (itimes-list (eval$ 'list bag a)))
         (y (itimes-list (eval$ 'list (bagdiff x bag) a)))
         (z (itimes-list (eval$ 'list (bagdiff y bag) a))))
        (itimes-list-bagdiff (y x) (x bag) (a a))
        (itimes-list-bagdiff (y y) (x bag) (a a)))))


(disable ilessp-trichotomy)


(disable same-fix-int-implies-not-ilessp)


(prove-lemma correctness-of-cancel-itimes-ilessp ((meta ilessp))
       (equal (eval$ t x a)
              (eval$ t (cancel-itimes-ilessp x) a))
       ((do-not-induct t)))

;; I think that the following lemma is safe because it won't be
;; called at all during relieve-hyps.


(prove-lemma ilessp-strict (rewrite)
  (implies (ilessp x y)
           (not (ilessp y x)))
  ((enable-theory integer-defns)))


; ---------- Setting up the State ----------

;; I'll close by disabling (or enabling) those rules and definitions
;; whose status as left over from above isn't quite what I'd like.
;; I'm going to leave the eval$ rules on and eval$ off.


(disable eval$-cancel-iplus)

(disable eval$-iplus)


(disable lessp-count-listp-cdr)
```

```
(disable eval$-iplus-tree-rec)

(disable eval$-iplus-tree)
;;(disable eval$-list-append) ;; Nice rule -- I'll keep it enabled

(disable iplus-list-eval$-fringe)

(disable eval$-iplus-list-bagdiff)

(disable lessp-difference-plus-arg1)

(disable lessp-difference-plus-arg1-commuted)

(disable correctness-of-cancel-iplus-ilessp-lemma)

(disable eval$-ilessp-iplus-tree-no-fix-int)

(disable make-cancel-iplus-inequality-simplifier)

(disable quotient-difference-lessp-arg2)

(disable eval$-itimes-tree-rec)

(disable eval$-itimes-tree)

(disable itimes-list-eval$-fringe)

(disable integerp-eval$-itimes)

(disable itimes-list-bagdiff)

(disable equal-itimes-list-eval$-list-delete)

(disable member-izerop-itimes-fringe)

(disable correctness-of-cancel-itimes-hack-1)

(disable eval$-make-cancel-itimes-equality)

(disable eval$-make-cancel-itimes-equality-1)

(disable eval$-make-cancel-itimes-equality-2)

(disable eval$-equal-itimes-tree-itimes-fringe-0)

(disable izerop-eval-of-member-implies-itimes-list-0)

(disable subsetp-implies-itimes-list-eval$-equals-0)

(disable equal-0-itimes-list-eval$-bagint-1)

(disable equal-0-itimes-list-eval$-bagint-2)

(disable correctness-of-cancel-itimes-hack-2)
```

```
(disable correctness-of-cancel-itimes-hack-3-lemma)

(disable correctness-of-cancel-itimes-hack-3)

(disable eval$-itimes-tree-no-fix-int-1)

(disable eval$-itimes-tree-no-fix-int-2)

(disable eval$-make-cancel-itimes-inequality)

(disable listp-bagint-with-singleton-implies-member)

(disable itimes-list-eval$-list-0)

(disable correctness-of-cancel-itimes-ilessp-hack-1)

(disable listp-bagint-with-singleton-member)

(disable correctness-of-cancel-itimes-ilessp-hack-2)

(disable correctness-of-cancel-itimes-ilessp-hack-3-lemma-1)

(disable correctness-of-cancel-itimes-ilessp-hack-3-lemma-2)

(disable correctness-of-cancel-itimes-ilessp-hack-3)

(disable correctness-of-cancel-itimes-ilessp-hack-4)
;; The last one is a tough call, but I think it's OK.
;; (disable ilessp-strict)

;;;;;; ***** EXTRA META STUFF ***** ;;;;;;

;; The next goal is to improve itimes cancellation so that it looks
;; for common factors, and hence works on equations like
;;    x*y + x = x*z
;; and, for that matter,
;; a*x + -b*x = 0.

;; Rather than changing the existing cancel-itimes function, I'll
;; leave that one but disable its metalemma at the end.  Then if the
;; new version, which I'll call cancel-itimes-factors, is found to be
;; too slow, one can always disable its metalemma and re-enable the
;; metalemma for cancel-itimes.

;; Notice, by the way, that the existing cancel-itimes function is
;; useless for something like the following, since there's no special
;; treatment for INEG.  I'll remedy that in this version.

#|
```

```
(IMPLIES (AND (NOT (IZEROP X))
              (EQUAL (ITIMES A X) (INEG (ITIMES B X))))
         (EQUAL (FIX-INT A) (INEG B)))
|#


(defn itimes-tree-ineg (l)

  ;; Like itimes-tree-rec in that it doesn't apply fix-int even for a
  ;; one-element list, but with special treatment if l is a list
  ;; starting with (quote -1).  Notice the coding with IF, for
  ;; computational efficiency.

  (if (listp l)
      (if (equal (car l) (list 'quote -1))
          (if (listp (cdr l))
              (list 'ineg (itimes-tree-rec (cdr l)))
            (car l))
        (itimes-tree-rec l))
    (quote (quote 1))))


(defn itimes-factors (x)
  ;; a "generalization" of itimes-fringe
  (if (listp x)
      (cond
       ((equal (car x)
               (quote itimes))
        (append (itimes-factors (cadr x))
                (itimes-factors (caddr x))))
       ((equal (car x)
               (quote iplus))
        (let ((bag1 (itimes-factors (cadr x)))
              (bag2 (itimes-factors (caddr x))))
          (let ((inboth (bagint bag1 bag2)))
            (if (listp inboth)
                (cons (list 'iplus
                            (itimes-tree-ineg (bagdiff bag1 inboth))
                            (itimes-tree-ineg (bagdiff bag2 inboth)))
                      inboth)
              (list x)))))
       ((equal (car x)
               (quote ineg))
        (cons (list 'quote -1)
              (itimes-factors (cadr x))))
       (t
        (list x)))
    (list x)))
```

```
(prove-lemma itimes--1 (rewrite)
             (equal (itimes -1 x)
                    (ineg x))
             ((enable-theory integer-defns)))

;; I'll need the following lemma because it's simplest not to deal with
;; e.g. (equal x x), where x is a variable, in the meta thing.  I'll do
;; the one after it too, simply because I'm thinking of it now.

(prove-lemma equal-ineg-ineg (rewrite)
  (equal (equal (ineg x) (ineg y))
         (equal (fix-int x) (fix-int y)))
  ((enable-theory integer-defns)))


(prove-lemma ilessp-ineg-ineg (rewrite)
  (equal (ilessp (ineg x) (ineg y))
         (ilessp y x))
  ((enable-theory integer-defns)))


(prove-lemma fix-int-eval$-itimes-tree-rec (rewrite)
  (implies (listp x)
           (equal (fix-int (eval$ t (itimes-tree-rec x) a))
                  (itimes-list (eval$ 'list x a))))
  ((enable eval$-itimes-tree-rec)))


(prove-lemma eval$-itimes-tree-ineg (rewrite)
  (equal (fix-int (eval$ t (itimes-tree-ineg x) a))
         (itimes-list (eval$ 'list x a)))
  ((enable eval$-itimes-tree-rec)))

;; Now I want the above lemma to apply, but it doesn't, so the
;; following three lemmas are used instead.


(prove-lemma ineg-eval$-itimes-tree-ineg (rewrite)
  (equal (ineg (eval$ t (itimes-tree-ineg x) a))
         (ineg (itimes-list (eval$ 'list x a))))
  ((use (eval$-itimes-tree-ineg))))


(prove-lemma iplus-eval$-itimes-tree-ineg (rewrite)
  (and (equal (iplus (eval$ t (itimes-tree-ineg x) a) y)
              (iplus (itimes-list (eval$ 'list x a)) y))
       (equal (iplus y (eval$ t (itimes-tree-ineg x) a))
              (iplus y (itimes-list (eval$ 'list x a)))))
  ((use (eval$-itimes-tree-ineg))))
```

```
(prove-lemma itimes-eval$-itimes-tree-ineg (rewrite)
  (and (equal (itimes (eval$ t (itimes-tree-ineg x) a) y)
              (itimes (itimes-list (eval$ 'list x a)) y))
       (equal (itimes y (eval$ t (itimes-tree-ineg x) a))
              (itimes y (itimes-list (eval$ 'list x a)))))
  ((use (eval$-itimes-tree-ineg))))


(disable itimes-tree-ineg)

#| ****** The following definitions are for efficient execution of
   metafunctions.  They should probably be applied to all the metafunctions
   with fns arguments AND and OR.


(defmacro nqthm-macroexpand (defn &rest fns)
  `(nqthm-macroexpand-fn ',defn ',fns))


(defun nqthm-macroexpand-fn (defn fns)
  (iterate for fn in fns
           when (not (get fn 'sdefn))
           do (er soft (fn) |Sorry| |,| |but| |there| |is| |no| SDEFN
                  |for| (!ppr fn (quote |.|))))
  (let (name args body)
    (match! defn (defn name args body))
    (let ((arity-alist (cons (cons name (length args)) arity-alist)))
      (list 'defn name args
            (untranslate (normalize-ifs
                          (nqthm-macroexpand-term (translate body) fns)
                          nil nil nil))))))
```

```
(defun nqthm-macroexpand-term (term fns)
  (cond
   ((or (variablep term) (fquotep term))
    term)
   ((member-eq (ffn-symb term) fns)
    (let ((sdefn (get (ffn-symb term) 'sdefn)))
      (sub-pair-var (cadr sdefn)
                    (iterate for arg in (fargs term)
                             collect (nqthm-macroexpand-term arg fns))
                    (caddr sdefn))))
   (t (fcons-term (ffn-symb term)
                  (iterate for arg in (fargs term)
                           collect (nqthm-macroexpand-term arg fns))))))

|#

;; I "macroexpand" away the following below, so it's not really needed
;; except for the proof.  That is, I use it in the definition of
;; cancel-itimes-factors, but then get rid of it for
;; cancel-itimes-factors-expanded, and although I reason about the
;; former, I USE the latter, for efficiency.


(defn iplus-or-itimes-term (x)
  (if (listp x)
      (case (car x)
            (iplus t)
            (itimes t)
            (ineg (if (listp (cadr x))
                      (equal (car (cadr x)) 'itimes)
                    f))
            (otherwise f))
    f))
```

```
(defn cancel-itimes-factors (x)
  (if (and (listp x)
           (equal (car x) (quote equal)))
      (let ((bagint (bagint (itimes-factors (cadr x))
                            (itimes-factors (caddr x)))))
        (let ((new-equality
               (make-cancel-itimes-equality (itimes-factors (cadr x))
                                            (itimes-factors (caddr x))
                                            bagint)))
          (if (iplus-or-itimes-term (cadr x))
              (if (listp bagint)
                  (if (iplus-or-itimes-term (caddr x))
                      new-equality
                    (list 'if
                          (list 'integerp (caddr x))
                          new-equality
                          (list 'quote f)))
                x)
            (if (iplus-or-itimes-term (caddr x))
                (if (listp bagint)
                    (list 'if
                          (list 'integerp (cadr x))
                          new-equality
                          (list 'quote f))
                  x)
              x))))
    x))

;; The following was generated with the nqthm-macroexpand macro defined above.
```

```
(DEFN CANCEL-ITIMES-FACTORS-expanded (X)
  (IF (LISTP X)
      (IF (EQUAL (CAR X) 'EQUAL)
          (COND
           ((LISTP (CADR X))
            (CASE
             (CAR (CAR (CDR X)))
             (IPLUS
              (IF (LISTP (BAGINT (ITIMES-FACTORS (CADR X))
                                 (ITIMES-FACTORS (CADDR X))))
                  (IF (LISTP (CADDR X))
                      (CASE (CAR (CAR (CDR (CDR X))))
                            (IPLUS
                             (MAKE-CANCEL-ITIMES-EQUALITY
                              (ITIMES-FACTORS (CADR X))
                              (ITIMES-FACTORS (CADDR X))
                              (BAGINT (ITIMES-FACTORS (CADR X))
                                      (ITIMES-FACTORS (CADDR X)))))
                            (ITIMES
                             (MAKE-CANCEL-ITIMES-EQUALITY
                              (ITIMES-FACTORS (CADR X))
                              (ITIMES-FACTORS (CADDR X))
                              (BAGINT (ITIMES-FACTORS (CADR X))
                                      (ITIMES-FACTORS (CADDR X)))))
                            (INEG
                             (IF (LISTP (CADADDR X))
                                 (IF (EQUAL (CAADADDR X) 'ITIMES)
                                     (MAKE-CANCEL-ITIMES-EQUALITY
                                      (ITIMES-FACTORS (CADR X))
                                      (ITIMES-FACTORS (CADDR X))
                                      (BAGINT
                                       (ITIMES-FACTORS (CADR X))
                                       (ITIMES-FACTORS (CADDR X))))
                                     (LIST 'IF
                                           (LIST 'INTEGERP (CADDR X))
                                           (MAKE-CANCEL-ITIMES-EQUALITY
                                            (ITIMES-FACTORS (CADR X))
                                            (ITIMES-FACTORS (CADDR X))
                                            (BAGINT
                                             (ITIMES-FACTORS (CADR X))
                                             (ITIMES-FACTORS (CADDR X))))
                                           (LIST 'QUOTE F)))
                                 (LIST 'IF
                                       (LIST 'INTEGERP (CADDR X))
                                       (MAKE-CANCEL-ITIMES-EQUALITY
                                        (ITIMES-FACTORS (CADR X))
                                        (ITIMES-FACTORS (CADDR X))
                                        (BAGINT
                                         (ITIMES-FACTORS (CADR X))
                                         (ITIMES-FACTORS (CADDR X))))
                                       (LIST 'QUOTE F))))
                            (OTHERWISE
                             (LIST 'IF
                                   (LIST 'INTEGERP (CADDR X))
```

```
                      (MAKE-CANCEL-ITIMES-EQUALITY
                       (ITIMES-FACTORS (CADR X))
                       (ITIMES-FACTORS (CADDR X))
                       (BAGINT
                        (ITIMES-FACTORS (CADR X))
                        (ITIMES-FACTORS (CADDR X))))
                      (LIST 'QUOTE F))))
         (LIST 'IF (LIST 'INTEGERP (CADDR X))
               (MAKE-CANCEL-ITIMES-EQUALITY
                (ITIMES-FACTORS (CADR X))
                (ITIMES-FACTORS (CADDR X))
                (BAGINT
                 (ITIMES-FACTORS (CADR X))
                 (ITIMES-FACTORS (CADDR X))))
               (LIST 'QUOTE F)))
     X))
(ITIMES
 (IF (LISTP (BAGINT (ITIMES-FACTORS (CADR X))
                    (ITIMES-FACTORS (CADDR X))))
     (IF (LISTP (CADDR X))
         (CASE (CAR (CAR (CDR (CDR X))))
               (IPLUS
                (MAKE-CANCEL-ITIMES-EQUALITY
                 (ITIMES-FACTORS (CADR X))
                 (ITIMES-FACTORS (CADDR X))
                 (BAGINT
                  (ITIMES-FACTORS (CADR X))
                  (ITIMES-FACTORS (CADDR X)))))
               (ITIMES
                (MAKE-CANCEL-ITIMES-EQUALITY
                 (ITIMES-FACTORS (CADR X))
                 (ITIMES-FACTORS (CADDR X))
                 (BAGINT
                  (ITIMES-FACTORS (CADR X))
                  (ITIMES-FACTORS (CADDR X)))))
               (INEG
                (IF (LISTP (CADADDR X))
                    (IF (EQUAL (CAADADDR X) 'ITIMES)
                        (MAKE-CANCEL-ITIMES-EQUALITY
                         (ITIMES-FACTORS (CADR X))
                         (ITIMES-FACTORS (CADDR X))
                         (BAGINT
                          (ITIMES-FACTORS (CADR X))
                          (ITIMES-FACTORS (CADDR X))))
                        (LIST 'IF
                              (LIST 'INTEGERP (CADDR X))
                              (MAKE-CANCEL-ITIMES-EQUALITY
                               (ITIMES-FACTORS (CADR X))
                               (ITIMES-FACTORS (CADDR X))
                               (BAGINT
                                (ITIMES-FACTORS (CADR X))
                                (ITIMES-FACTORS (CADDR X))))
                              (LIST 'QUOTE F)))
                    (LIST 'IF
                          (LIST 'INTEGERP (CADDR X))
```

```
                              (MAKE-CANCEL-ITIMES-EQUALITY
                               (ITIMES-FACTORS (CADR X))
                               (ITIMES-FACTORS (CADDR X))
                               (BAGINT
                                (ITIMES-FACTORS (CADR X))
                                (ITIMES-FACTORS (CADDR X))))
                              (LIST 'QUOTE F))))
                   (OTHERWISE
                    (LIST 'IF
                          (LIST 'INTEGERP (CADDR X))
                          (MAKE-CANCEL-ITIMES-EQUALITY
                           (ITIMES-FACTORS (CADR X))
                           (ITIMES-FACTORS (CADDR X))
                           (BAGINT
                            (ITIMES-FACTORS (CADR X))
                            (ITIMES-FACTORS (CADDR X))))
                          (LIST 'QUOTE F))))
             (LIST 'IF (LIST 'INTEGERP (CADDR X))
                   (MAKE-CANCEL-ITIMES-EQUALITY
                    (ITIMES-FACTORS (CADR X))
                    (ITIMES-FACTORS (CADDR X))
                    (BAGINT (ITIMES-FACTORS (CADR X))
                            (ITIMES-FACTORS (CADDR X))))
                   (LIST 'QUOTE F)))
      X))
 (INEG (COND
        ((LISTP (CADADR X))
         (COND
          ((EQUAL (CAADADR X) 'ITIMES)
           (IF (LISTP
                (BAGINT (ITIMES-FACTORS (CADR X))
                        (ITIMES-FACTORS (CADDR X))))
               (IF (LISTP (CADDR X))
                   (CASE (CAR (CAR (CDR (CDR X))))
                         (IPLUS
                          (MAKE-CANCEL-ITIMES-EQUALITY
                           (ITIMES-FACTORS (CADR X))
                           (ITIMES-FACTORS (CADDR X))
                           (BAGINT
                            (ITIMES-FACTORS (CADR X))
                            (ITIMES-FACTORS (CADDR X)))))
                         (ITIMES
                          (MAKE-CANCEL-ITIMES-EQUALITY
                           (ITIMES-FACTORS (CADR X))
                           (ITIMES-FACTORS (CADDR X))
                           (BAGINT
                            (ITIMES-FACTORS (CADR X))
                            (ITIMES-FACTORS (CADDR X)))))
                         (INEG
                          (IF (LISTP (CADADDR X))
                              (IF
                               (EQUAL (CAADADDR X) 'ITIMES)
                               (MAKE-CANCEL-ITIMES-EQUALITY
                                (ITIMES-FACTORS (CADR X))
                                (ITIMES-FACTORS (CADDR X))
```

```
                              (BAGINT
                               (ITIMES-FACTORS (CADR X))
                               (ITIMES-FACTORS (CADDR X))))
                             (LIST 'IF
                                   (LIST 'INTEGERP (CADDR X))
                                   (MAKE-CANCEL-ITIMES-EQUALITY
                                    (ITIMES-FACTORS (CADR X))
                                    (ITIMES-FACTORS (CADDR X))
                                    (BAGINT
                                     (ITIMES-FACTORS (CADR X))
                                     (ITIMES-FACTORS (CADDR X))))
                                   (LIST 'QUOTE F)))
                            (LIST 'IF
                                  (LIST 'INTEGERP (CADDR X))
                                  (MAKE-CANCEL-ITIMES-EQUALITY
                                   (ITIMES-FACTORS (CADR X))
                                   (ITIMES-FACTORS (CADDR X))
                                   (BAGINT
                                    (ITIMES-FACTORS (CADR X))
                                    (ITIMES-FACTORS (CADDR X))))
                                  (LIST 'QUOTE F))))
                       (OTHERWISE
                        (LIST 'IF
                              (LIST 'INTEGERP (CADDR X))
                              (MAKE-CANCEL-ITIMES-EQUALITY
                               (ITIMES-FACTORS (CADR X))
                               (ITIMES-FACTORS (CADDR X))
                               (BAGINT
                                (ITIMES-FACTORS (CADR X))
                                (ITIMES-FACTORS (CADDR X))))
                              (LIST 'QUOTE F))))
                 (LIST 'IF
                       (LIST 'INTEGERP (CADDR X))
                       (MAKE-CANCEL-ITIMES-EQUALITY
                        (ITIMES-FACTORS (CADR X))
                        (ITIMES-FACTORS (CADDR X))
                        (BAGINT (ITIMES-FACTORS (CADR X))
                                (ITIMES-FACTORS (CADDR X))))
                       (LIST 'QUOTE F)))
             X))
        ((LISTP (CADDR X))
         (CASE (CAR (CAR (CDR (CDR X))))
               (IPLUS (IF
                       (LISTP
                        (BAGINT
                         (ITIMES-FACTORS (CADR X))
                         (ITIMES-FACTORS (CADDR X))))
                       (LIST 'IF
                             (LIST 'INTEGERP (CADR X))
                             (MAKE-CANCEL-ITIMES-EQUALITY
                              (ITIMES-FACTORS (CADR X))
                              (ITIMES-FACTORS (CADDR X))
                              (BAGINT
                               (ITIMES-FACTORS (CADR X))
                               (ITIMES-FACTORS (CADDR X))))
```

```
                         (LIST 'QUOTE F))
                   X))
          (ITIMES (IF
                   (LISTP
                    (BAGINT
                     (ITIMES-FACTORS (CADR X))
                     (ITIMES-FACTORS (CADDR X))))
                   (LIST 'IF
                         (LIST 'INTEGERP (CADR X))
                         (MAKE-CANCEL-ITIMES-EQUALITY
                          (ITIMES-FACTORS (CADR X))
                          (ITIMES-FACTORS (CADDR X))
                          (BAGINT
                           (ITIMES-FACTORS (CADR X))
                           (ITIMES-FACTORS (CADDR X))))
                         (LIST 'QUOTE F))
                   X))
          (INEG (IF (LISTP (CADADDR X))
                    (IF
                     (EQUAL (CAADADDR X) 'ITIMES)
                     (IF
                      (LISTP
                       (BAGINT
                        (ITIMES-FACTORS (CADR X))
                        (ITIMES-FACTORS (CADDR X))))
                      (LIST 'IF
                            (LIST 'INTEGERP (CADR X))
                            (MAKE-CANCEL-ITIMES-EQUALITY
                             (ITIMES-FACTORS (CADR X))
                             (ITIMES-FACTORS (CADDR X))
                             (BAGINT
                              (ITIMES-FACTORS (CADR X))
                              (ITIMES-FACTORS (CADDR X))))
                            (LIST 'QUOTE F))
                      X)
                     X)
                    X))
          (OTHERWISE X)))
   (T X)))
 ((LISTP (CADDR X))
  (CASE (CAR (CAR (CDR (CDR X))))
        (IPLUS (IF
                (LISTP
                 (BAGINT
                  (ITIMES-FACTORS (CADR X))
                  (ITIMES-FACTORS (CADDR X))))
                (LIST 'IF
                      (LIST 'INTEGERP (CADR X))
                      (MAKE-CANCEL-ITIMES-EQUALITY
                       (ITIMES-FACTORS (CADR X))
                       (ITIMES-FACTORS (CADDR X))
                       (BAGINT
                        (ITIMES-FACTORS (CADR X))
                        (ITIMES-FACTORS (CADDR X))))
                      (LIST 'QUOTE F))
```

```
                        X))
                (ITIMES (IF
                         (LISTP
                          (BAGINT
                           (ITIMES-FACTORS (CADR X))
                           (ITIMES-FACTORS (CADDR X))))
                         (LIST 'IF
                               (LIST 'INTEGERP (CADR X))
                               (MAKE-CANCEL-ITIMES-EQUALITY
                                (ITIMES-FACTORS (CADR X))
                                (ITIMES-FACTORS (CADDR X))
                                (BAGINT
                                 (ITIMES-FACTORS (CADR X))
                                 (ITIMES-FACTORS (CADDR X))))
                               (LIST 'QUOTE F))
                         X))
                (INEG (IF (LISTP (CADADDR X))
                          (IF (EQUAL (CAADADDR X) 'ITIMES)
                              (IF
                               (LISTP
                                (BAGINT
                                 (ITIMES-FACTORS (CADR X))
                                 (ITIMES-FACTORS (CADDR X))))
                               (LIST 'IF
                                     (LIST 'INTEGERP (CADR X))
                                     (MAKE-CANCEL-ITIMES-EQUALITY
                                      (ITIMES-FACTORS (CADR X))
                                      (ITIMES-FACTORS (CADDR X))
                                      (BAGINT
                                       (ITIMES-FACTORS (CADR X))
                                       (ITIMES-FACTORS (CADDR X))))
                                     (LIST 'QUOTE F))
                               X)
                              X)
                          X))
                (OTHERWISE X)))
         (T X)))
  (OTHERWISE
   (IF (LISTP (CADDR X))
       (CASE (CAR (CAR (CDR (CDR X))))
             (IPLUS (IF
                     (LISTP
                      (BAGINT (ITIMES-FACTORS (CADR X))
                              (ITIMES-FACTORS (CADDR X))))
                     (LIST 'IF
                           (LIST 'INTEGERP (CADR X))
                           (MAKE-CANCEL-ITIMES-EQUALITY
                            (ITIMES-FACTORS (CADR X))
                            (ITIMES-FACTORS (CADDR X))
                            (BAGINT
                             (ITIMES-FACTORS (CADR X))
                             (ITIMES-FACTORS (CADDR X))))
                           (LIST 'QUOTE F))
                     X))
             (ITIMES (IF
```

```
                         (LISTP
                          (BAGINT
                           (ITIMES-FACTORS (CADR X))
                           (ITIMES-FACTORS (CADDR X))))
                         (LIST 'IF
                               (LIST 'INTEGERP (CADR X))
                               (MAKE-CANCEL-ITIMES-EQUALITY
                                (ITIMES-FACTORS (CADR X))
                                (ITIMES-FACTORS (CADDR X))
                                (BAGINT
                                 (ITIMES-FACTORS (CADR X))
                                 (ITIMES-FACTORS (CADDR X))))
                               (LIST 'QUOTE F))
                         X))
                (INEG (IF (LISTP (CADADDR X))
                          (IF (EQUAL (CAADADDR X) 'ITIMES)
                              (IF
                               (LISTP
                                (BAGINT
                                 (ITIMES-FACTORS (CADR X))
                                 (ITIMES-FACTORS (CADDR X))))
                               (LIST 'IF
                                     (LIST 'INTEGERP (CADR X))
                                     (MAKE-CANCEL-ITIMES-EQUALITY
                                      (ITIMES-FACTORS (CADR X))
                                      (ITIMES-FACTORS (CADDR X))
                                      (BAGINT
                                       (ITIMES-FACTORS (CADR X))
                                       (ITIMES-FACTORS (CADDR X))))
                                     (LIST 'QUOTE F))
                               X)
                              X)
                          X))
                (OTHERWISE X))
         X))))
  ((LISTP (CADDR X))
   (CASE (CAR (CAR (CDR (CDR X))))
         (IPLUS (IF (LISTP (BAGINT (ITIMES-FACTORS (CADR X))
                                   (ITIMES-FACTORS (CADDR X))))
                    (LIST 'IF (LIST 'INTEGERP (CADR X))
                          (MAKE-CANCEL-ITIMES-EQUALITY
                           (ITIMES-FACTORS (CADR X))
                           (ITIMES-FACTORS (CADDR X))
                           (BAGINT (ITIMES-FACTORS (CADR X))
                                   (ITIMES-FACTORS (CADDR X))))
                          (LIST 'QUOTE F))
                    X))
         (ITIMES (IF (LISTP (BAGINT (ITIMES-FACTORS (CADR X))
                                    (ITIMES-FACTORS (CADDR X))))
                     (LIST 'IF (LIST 'INTEGERP (CADR X))
                           (MAKE-CANCEL-ITIMES-EQUALITY
                            (ITIMES-FACTORS (CADR X))
                            (ITIMES-FACTORS (CADDR X))
                            (BAGINT (ITIMES-FACTORS (CADR X))
                                    (ITIMES-FACTORS (CADDR X))))
```

```
                                (LIST 'QUOTE F))
                          X))
                  (INEG (IF (LISTP (CADADDR X))
                            (IF (EQUAL (CAADADDR X) 'ITIMES)
                                (IF (LISTP
                                     (BAGINT (ITIMES-FACTORS (CADR X))
                                             (ITIMES-FACTORS (CADDR X))))
                                    (LIST 'IF
                                          (LIST 'INTEGERP (CADR X))
                                          (MAKE-CANCEL-ITIMES-EQUALITY
                                           (ITIMES-FACTORS (CADR X))
                                           (ITIMES-FACTORS (CADDR X))
                                           (BAGINT
                                            (ITIMES-FACTORS (CADR X))
                                            (ITIMES-FACTORS (CADDR X))))
                                          (LIST 'QUOTE F))
                                    X)
                                X)
                            X))
                  (OTHERWISE X)))
           (T X))
          X)
      X))


(prove-lemma cancel-itimes-factors-expanded-cancel-itimes-factors (rewrite)
             (equal (cancel-itimes-factors-expanded x)
                    (cancel-itimes-factors x))
             ((disable-theory t)
              (enable-theory ground-zero)
              (enable iplus-or-itimes-term cancel-itimes-factors
                      cancel-itimes-factors-expanded)))


(disable cancel-itimes-factors-expanded)


(disable iplus-or-itimes-term)


(prove-lemma equal-itimes-list-eval$-list-delete-new-1 (rewrite)
  (implies (not (equal (fix-int (eval$ t elt a)) 0))
           (equal (equal x (itimes-list (eval$ 'list
                                               (delete elt bag)
                                               a)))
                  (if (member elt bag)
                      (and (integerp x)
                           (equal (itimes x (eval$ t elt a) )
                                  (itimes-list (eval$ 'list bag a))))
                    (equal x (itimes-list (eval$ 'list bag a))))))
  ((enable equal-itimes-list-eval$-list-delete)))
```

```
(prove-lemma equal-itimes-list-eval$-list-delete-new-2 (rewrite)
  (implies (not (equal (fix-int (eval$ t elt a)) 0))
           (equal (equal (itimes-list (eval$ 'list
                                             (delete elt bag)
                                             a))
                         x)
                  (if (member elt bag)
                      (and (integerp x)
                           (equal (itimes x (eval$ t elt a) )
                                  (itimes-list (eval$ 'list bag a))))
                    (equal x (itimes-list (eval$ 'list bag a)))))))


(prove-lemma itimes-itimes-list-eval$-list-delete (rewrite)
  (implies (member x bag)
           (equal (itimes (eval$ t x a)
                          (itimes-list (eval$ 'list (delete x bag) a)))
                  (itimes-list (eval$ 'list bag a)))))


(prove-lemma equal-itimes-list-eval$-list-bagdiff (rewrite)
  (implies (and (subbagp in-both bag1)
                (subbagp in-both bag2)
                (not (equal (itimes-list (eval$ 'list in-both a)) 0)))
           (equal (equal (itimes-list (eval$ 'list
                                             (bagdiff bag1 in-both)
                                             a))
                         (itimes-list (eval$ 'list
                                             (bagdiff bag2 in-both)
                                             a)))
                  (equal (itimes-list (eval$ 'list bag1 a))
                         (itimes-list (eval$ 'list bag2 a))))))


(prove-lemma membership-of-0-implies-itimes-list-is-0 (rewrite)
  (implies (member 0 x)
           (equal (itimes-list x) 0)))


(prove-lemma member-0-eval$-list (rewrite)
  (implies (member 0 x)
           (member 0 (eval$ 'list x a))))
```

```
(prove-lemma itimes-list-eval$-factors-lemma (rewrite)
  (equal (itimes (itimes-list (eval$ 'list
                                    (bagint bag1 bag2)
                                    a))
                 (itimes-list (eval$ 'list
                                    (bagdiff bag2 (bagint bag1 bag2))
                                    a)))
         (itimes-list (eval$ 'list bag2 a)))
  ((use (itimes-list-bagdiff
         (x (bagint bag1 bag2))
         (y bag2)))))


(prove-lemma itimes-list-eval$-factors-lemma-prime (rewrite)
  (equal (itimes (itimes-list (eval$ 'list
                                    (bagint bag1 bag2)
                                    a))
                 (itimes-list (eval$ 'list
                                    (bagdiff bag1 (bagint bag1 bag2))
                                    a)))
         (itimes-list (eval$ 'list bag1 a)))
  ((use (itimes-list-bagdiff
         (x (bagint bag1 bag2))
         (y bag1)))))


(prove-lemma itimes-list-eval$-factors (rewrite)
       ;; similar to ITIMES-LIST-EVAL$-FRINGE, except one has to
       ;; reason about bagdiff etc.
       (equal (itimes-list (eval$ 'list (itimes-factors x) a))
              (fix-int (eval$ t x a)))
       ((induct (itimes-factors x))
        (enable ;;eval$-list-append ;; already enabled now
                integerp-eval$-itimes
                itimes-list-bagdiff
                listp-bagint-with-singleton-member)))


(prove-lemma iplus-or-itimes-term-integerp-eval$ (rewrite)
  (implies (iplus-or-itimes-term x)
           (integerp (eval$ t x a)))
  ((enable iplus-or-itimes-term)))
```

```
(prove-lemma eval$-list-bagint-0 nil
  (implies (equal (itimes-list (eval$ 'list (bagint x y) a)) 0)
           (and (equal (itimes-list (eval$ 'list x a)) 0)
                (equal (itimes-list (eval$ 'list y a)) 0)))
  ((use (subsetp-implies-itimes-list-eval$-equals-0
         (x (bagint x y))
         (y x))
        (subsetp-implies-itimes-list-eval$-equals-0
         (x (bagint x y))
         (y y)))))


(prove-lemma eval$-list-bagint-0-implies-equal (rewrite)
  (implies (and (equal (itimes-list
                        (eval$ 'list (bagint (itimes-factors v)
                                             (itimes-factors w)) a))
                       0)
                (integerp (eval$ t v a))
                (integerp (eval$ t w a)))
           (equal (equal (eval$ t v a) (eval$ t w a))
                  t))
  ((use (eval$-list-bagint-0 (x (itimes-factors v))
                             (y (itimes-factors w))))))


(prove-lemma correctness-of-cancel-itimes-factors ((meta equal))
       (equal (eval$ t x a)
              (eval$ t (cancel-itimes-factors-expanded x) a))
       ((do-not-induct t)
        (enable eval$-itimes-tree eval$-make-cancel-itimes-equality)))

;; OK -- now, the lessp case, finally.  Ugh!


(defn cancel-itimes-ilessp-factors (x)
  (if (listp x)
      (if (equal (car x) 'ilessp)
          (if (listp (bagint (itimes-factors (cadr x))
                             (itimes-factors (caddr x))))
              (make-cancel-itimes-inequality
               (itimes-factors (cadr x))
               (itimes-factors (caddr x))
               (bagint (itimes-factors (cadr x))
                       (itimes-factors (caddr x))))
              x)
          x)
      x))
```

```
(prove-lemma bagint-singleton (rewrite)
  (equal (bagint x (list y))
         (if (member y x)
             (list y)
           nil)))


(prove-lemma izerop-ilessp-0-relationship (rewrite)
  (equal (equal (fix-int x) 0)
         (and (not (ilessp x 0))
              (not (ilessp 0 x))))
  ((enable-theory integer-defns)))


(prove-lemma ilessp-itimes-list-eval$-list-delete-helper-1 (rewrite)
  (implies
   (ilessp 0 w)
   (equal
    (ilessp (itimes x w)
            (itimes w u))
    (ilessp x u))))


(prove-lemma ilessp-itimes-list-eval$-list-delete-helper-2 (rewrite)
  (implies
   (ilessp w 0)
   (equal
    (ilessp (itimes w u)
            (itimes x w))
    (ilessp x u))))


(prove-lemma ilessp-itimes-list-eval$-list-delete (rewrite)
  (implies (and (member z y)
                (not (equal (fix-int (eval$ t z a)) 0)))
           (equal (ilessp x
                          (itimes-list (eval$ 'list (delete z y) a)))
                  (if (ilessp 0 (eval$ t z a))
                      (ilessp (itimes x (eval$ t z a))
                              (itimes-list (eval$ 'list y a)))
                    (if (ilessp (eval$ t z a) 0)
                        (ilessp (itimes-list (eval$ 'list y a))
                                (itimes x (eval$ t z a)))
                      f))))
  ((enable itimes-list-eval$-delete)
   (disable itimes-itimes-list-eval$-list-delete)))
```

```
(prove-lemma ilessp-itimes-list-eval$-list-delete-prime-helper-1 (rewrite)
  (implies
   (ilessp 0 w)
   (equal
    (ilessp (itimes w u)
            (itimes x w))
    (ilessp u x))))


(prove-lemma ilessp-itimes-list-eval$-list-delete-prime-helper-2 (rewrite)
  (implies
   (ilessp w 0)
   (equal
    (ilessp (itimes x w)
            (itimes w u))
    (ilessp u x))))


(prove-lemma ilessp-itimes-list-eval$-list-delete-prime (rewrite)
  (implies (and (member z y)
                (not (equal (fix-int (eval$ t z a)) 0)))
           (equal (ilessp (itimes-list (eval$ 'list (delete z y) a))
                          x)
                  (if (ilessp 0 (eval$ t z a))
                      (ilessp (itimes-list (eval$ 'list y a))
                              (itimes x (eval$ t z a)))
                    (if (ilessp (eval$ t z a) 0)
                        (ilessp (itimes x (eval$ t z a))
                                (itimes-list (eval$ 'list y a)))
                      f))))
  ((enable itimes-list-eval$-delete)
   (disable itimes-itimes-list-eval$-list-delete
            ilessp-itimes-list-eval$-list-delete)))

;; **** Do I have anything like the following two lemmas for the
;; equality case?  Should I?

;;;***** I should also consider if I've dealt with things like 0 = a*x
;;;+ b*x, and simlilarly for ilessp.


(prove-lemma ilessp-0-itimes (rewrite)
  (equal (ilessp 0 (itimes x y))
         (or (and (ilessp 0 x) (ilessp 0 y))
             (and (ilessp x 0) (ilessp y 0))))
  ((enable-theory integer-defns)))


(prove-lemma ilessp-itimes-0 (rewrite)
  (equal (ilessp (itimes x y) 0)
         (or (and (ilessp 0 x) (ilessp y 0))
             (and (ilessp x 0) (ilessp 0 y))))
  ((enable-theory integer-defns)))
```

```
(prove-lemma ilessp-itimes-list-eval$-list-bagdiff (rewrite)
  (implies (and (subbagp in-both bag1)
                (subbagp in-both bag2)
                (not (equal (itimes-list (eval$ 'list in-both a)) 0)))
           (equal (ilessp (itimes-list (eval$ 'list
                                              (bagdiff bag1 in-both)
                                              a))
                          (itimes-list (eval$ 'list
                                              (bagdiff bag2 in-both)
                                              a)))
                  (if (ilessp 0 (itimes-list (eval$ 'list in-both a)))
                      (ilessp (itimes-list (eval$ 'list bag1 a))
                              (itimes-list (eval$ 'list bag2 a)))
                    (ilessp (itimes-list (eval$ 'list bag2 a))
                            (itimes-list (eval$ 'list bag1 a))))))
  ((enable ilessp-trichotomy)
   (disable izerop-ilessp-0-relationship)))


(prove-lemma zero-ilessp-implies-not-equal nil
             ;; This is not a rewrite rule because I don't want to slow down
             ;; the rewriter.  Maybe that's not such a great decision.
             (implies (ilessp 0 x)
                      (not (equal 0 x))))


(prove-lemma ilessp-itimes-list-eval$-list-bagdiff-corollary-1 (rewrite)
  (implies (and (subbagp in-both bag1)
                (subbagp in-both bag2)
                (ilessp 0 (itimes-list (eval$ 'list in-both a))))
           (equal (ilessp (itimes-list (eval$ 'list
                                              (bagdiff bag1 in-both)
                                              a))
                          (itimes-list (eval$ 'list
                                              (bagdiff bag2 in-both)
                                              a)))
                  (ilessp (itimes-list (eval$ 'list bag1 a))
                          (itimes-list (eval$ 'list bag2 a)))))
  ((use (zero-ilessp-implies-not-equal (x (itimes-list
                                           (eval$ 'list in-both a)))))))


(prove-lemma ilessp-zero-implies-not-equal nil
             ;; This is not a rewrite rule because I don't want to slow down
             ;; the rewriter.  Maybe that's not such a great decision.
             (implies (ilessp x 0)
                      (not (equal 0 x))))
```

```
(prove-lemma ilessp-itimes-list-eval$-list-bagdiff-corollary-2 (rewrite)
  (implies (and (subbagp in-both bag1)
                (subbagp in-both bag2)
                (ilessp (itimes-list (eval$ 'list in-both a)) 0))
           (equal (ilessp (itimes-list (eval$ 'list
                                               (bagdiff bag1 in-both)
                                               a))
                          (itimes-list (eval$ 'list
                                               (bagdiff bag2 in-both)
                                               a)))
                  (ilessp (itimes-list (eval$ 'list bag2 a))
                          (itimes-list (eval$ 'list bag1 a))))))
  ((use (ilessp-zero-implies-not-equal (x (itimes-list
                                            (eval$ 'list in-both a)))))))


(prove-lemma member-0-itimes-factors-yields-0 (rewrite)
  ;; I'll hang this on MEMBER for efficiency
  (implies (not (equal (eval$ t w a) 0))
           (not (member 0 (itimes-factors w)))))


(prove-lemma member-0-itimes-factors-yields-0-ilessp-consequence-1 (rewrite)
  ;; I'll hang this on MEMBER for efficiency
  (implies (ilessp (eval$ t w a) 0)
           (not (member 0 (itimes-factors w))))
  ((use (member-0-itimes-factors-yields-0))))


(prove-lemma member-0-itimes-factors-yields-0-ilessp-consequence-2 (rewrite)
  ;; I'll hang this on MEMBER for efficiency
  (implies (ilessp 0 (eval$ t w a))
           (not (member 0 (itimes-factors w))))
  ((use (member-0-itimes-factors-yields-0))))

#|

(prove-lemma eval$-list-bagint-0 nil
  (implies (equal (itimes-list (eval$ 'list (bagint x y) a)) 0)
           (and (equal (itimes-list (eval$ 'list x a)) 0)
                (equal (itimes-list (eval$ 'list y a)) 0)))
  ((use (subsetp-implies-itimes-list-eval$-equals-0
         (x (bagint x y))
         (y x))
        (subsetp-implies-itimes-list-eval$-equals-0
         (x (bagint x y))
         (y y)))))
|#

#|
```

```
(prove-lemma eval$-list-bagint-0-implies-equal (rewrite)
  (implies (and (equal (itimes-list
                         (eval$ 'list (bagint (itimes-factors v)
                                              (itimes-factors w)) a))
                        0)
                (integerp (eval$ t v a))
                (integerp (eval$ t w a)))
           (equal (equal (eval$ t v a) (eval$ t w a))
                  t))
  ((use (eval$-list-bagint-0 (x (itimes-factors v))
                             (y (itimes-factors w))))))
|#

;; At this point I'm going to switch the states of ilessp-trichotomy and
;; izerop-ilessp-0-relationship, for good (or till I change my mind again!).


(enable ilessp-trichotomy)

(disable izerop-ilessp-0-relationship)


(prove-lemma eval$-list-bagint-0-for-ilessp nil
  (implies (and (not (ilessp (itimes-list (eval$ 'list (bagint x y) a)) 0))
                (not (ilessp 0 (itimes-list (eval$ 'list (bagint x y) a)))))
           (and (equal (fix-int (itimes-list (eval$ 'list x a))) 0)
                (equal (fix-int (itimes-list (eval$ 'list y a))) 0)))
  ((use (subsetp-implies-itimes-list-eval$-equals-0
         (x (bagint x y))
         (y x))
        (subsetp-implies-itimes-list-eval$-equals-0
         (x (bagint x y))
         (y y)))))


(prove-lemma eval$-list-bagint-0-implies-equal-for-ilessp-lemma nil
  (implies (and (not (ilessp (itimes-list (eval$ 'list
                                                 (bagint (itimes-factors v)
                                                         (itimes-factors w))
                                                 a))
                             0))
                (not (ilessp 0
                             (itimes-list (eval$ 'list
                                                 (bagint (itimes-factors v)
                                                         (itimes-factors w))
                                                 a)))))
           (equal (fix-int (eval$ t v a))
                  (fix-int (eval$ t w a))))
  ((use (eval$-list-bagint-0-for-ilessp (x (itimes-factors v))
                                        (y (itimes-factors w))))))
```

```
(prove-lemma equal-fix-int-to-ilessp nil
  ;; Not a rewrite rule, for efficiency
  (implies (equal (fix-int x) (fix-int y))
           (not (ilessp x y)))
  ((enable-theory integer-defns)))


(prove-lemma eval$-list-bagint-0-implies-equal-for-ilessp (rewrite)
  (implies (and (not (ilessp (itimes-list (eval$ 'list
                                                 (bagint (itimes-factors v)
                                                         (itimes-factors w))
                                                 a))
                             0))
                (not (ilessp 0
                             (itimes-list (eval$ 'list
                                                 (bagint (itimes-factors v)
                                                         (itimes-factors w))
                                                 a)))))
           (and (not (ilessp (eval$ t v a)
                             (eval$ t w a)))
                (not (ilessp (eval$ t w a)
                             (eval$ t v a)))))
  ((use (eval$-list-bagint-0-implies-equal-for-ilessp-lemma)
        (equal-fix-int-to-ilessp (x (eval$ t v a)) (y (eval$ t w a)))
        (equal-fix-int-to-ilessp (x (eval$ t w a)) (y (eval$ t v a))))))

;; The rewrite rule ILESSP-TRICHOTOMY seemed to mess up the proof of
;; the following, so I'm just going to leave it disabled.

(disable ilessp-trichotomy)


(prove-lemma correctness-of-cancel-itimes-ilessp-factors ((meta ilessp))
       (equal (eval$ t x a)
              (eval$ t (cancel-itimes-ilessp-factors x) a))
       ((do-not-induct t)
        (enable eval$-itimes-tree-no-fix-int-1
                eval$-itimes-tree-no-fix-int-2
                eval$-itimes-tree
                eval$-make-cancel-itimes-inequality)))

;; OK -- now, the zero cases.


(enable LESSP-COUNT-LISTP-CDR)


(defn disjoin-equalities-with-0 (factors)
  (if (listp (cdr factors))
      (list 'or
            (list 'equal (list 'fix-int (car factors)) ''0)
            (disjoin-equalities-with-0 (cdr factors)))
    (list 'equal (list 'fix-int (car factors)) ''0)))
```

```
(disable LESSP-COUNT-LISTP-CDR)


(defn cancel-factors-0 (x)
  (if (listp x)
      (if (equal (car x) 'equal)
          (if (equal (cadr x) ''0)
              (let ((factors (itimes-factors (caddr x))))
                (if (listp (cdr factors))
                    (disjoin-equalities-with-0 factors)
                  x))
            (if (equal (caddr x) ''0)
                (let ((factors (itimes-factors (cadr x))))
                  (if (listp (cdr factors))
                      (disjoin-equalities-with-0 factors)
                    x))
              x))
        x)
    x))


(defn some-eval$s-to-0 (x a)
  ;; says that some member of x eval$s to an izerop
  (if (listp x)
      (or (equal (fix-int (eval$ t (car x) a)) 0)
          (some-eval$s-to-0 (cdr x) a))
    f))


(prove-lemma eval$-disjoin-equalities-with-0 (rewrite)
  (implies (listp lst)
           (equal (eval$ t (disjoin-equalities-with-0 lst) a)
                  (some-eval$s-to-0 lst a))))


(prove-lemma some-eval$s-to-0-append (rewrite)
  (equal (some-eval$s-to-0 (append x y) a)
         (or (some-eval$s-to-0 x a)
             (some-eval$s-to-0 y a))))


(prove-lemma some-eval$s-to-0-eliminator (rewrite)
  (equal (some-eval$s-to-0 x a)
         (equal (itimes-list (eval$ 'list x a)) 0)))


(prove-lemma listp-cdr-factors-implies-integerp (rewrite)
  (implies (listp (cdr (itimes-factors v)))
           (integerp (eval$ t v a)))
  ((expand (itimes-factors v))))
```

```
(prove-lemma correctness-of-cancel-factors-0 ((meta equal))
  (equal (eval$ t x a)
         (eval$ t (cancel-factors-0 x) a)))

;; and now for inequalities...


(enable LESSP-COUNT-LISTP-CDR)


(defn conjoin-inequalities-with-0 (factors parity)
  ;; Returns an inequality saying that 0 is less than the product of the
  ;; factors if parity is not F and the other way around otherwise.
  (if (listp (cdr factors))
      (if parity
          (list 'or
                (list 'and
                      (list 'ilessp ''0 (car factors))
                      (conjoin-inequalities-with-0 (cdr factors) t))
                (list 'and
                      (list 'ilessp (car factors) ''0)
                      (conjoin-inequalities-with-0 (cdr factors) f)))
        (list 'or
              (list 'and
                    (list 'ilessp (car factors) ''0)
                    (conjoin-inequalities-with-0 (cdr factors) t))
              (list 'and
                    (list 'ilessp ''0 (car factors))
                    (conjoin-inequalities-with-0 (cdr factors) f))))
    (if parity
        (list 'ilessp ''0 (car factors))
      (list 'ilessp (car factors) ''0))))


(disable lessp-count-listp-cdr)


(defn cancel-factors-ilessp-0 (x)
  (if (listp x)
      (if (equal (car x) 'ilessp)
          (if (equal (cadr x) ''0)
              (let ((factors (itimes-factors (caddr x))))
                (if (listp (cdr factors))
                    (conjoin-inequalities-with-0 factors t)
                  x))
            (if (equal (caddr x) ''0)
                (let ((factors (itimes-factors (cadr x))))
                  (if (listp (cdr factors))
                      (conjoin-inequalities-with-0 factors f)
                    x))
              x))
        x)
    x))
```

```
(prove-lemma conjoin-inequalities-with-0-eliminator (rewrite)
  (implies (listp x)
           (equal (eval$ t (conjoin-inequalities-with-0 x parity) a)
                  (if parity
                      (ilessp 0 (itimes-list (eval$ 'list x a)))
                    (ilessp (itimes-list (eval$ 'list x a)) 0)))))


(prove-lemma correctness-of-cancel-factors-ilessp-0 ((meta ilessp))
  (equal (eval$ t x a)
         (eval$ t (cancel-factors-ilessp-0 x) a)))


(disable equal-itimes-list-eval$-list-delete-new-1)

(disable equal-itimes-list-eval$-list-delete-new-2)

(disable itimes-itimes-list-eval$-list-delete)

(disable equal-itimes-list-eval$-list-bagdiff)

(disable itimes-list-eval$-factors-lemma)

(disable itimes-list-eval$-factors-lemma-prime)

(disable itimes-list-eval$-factors)

(disable iplus-or-itimes-term-integerp-eval$)

(disable eval$-list-bagint-0)

(disable eval$-list-bagint-0-implies-equal)

(disable izerop-ilessp-0-relationship)

(disable ilessp-itimes-list-eval$-list-delete-helper-1)

(disable ilessp-itimes-list-eval$-list-delete-helper-2)

(disable ilessp-itimes-list-eval$-list-delete)

(disable ilessp-itimes-list-eval$-list-delete-prime-helper-1)

(disable ilessp-itimes-list-eval$-list-delete-prime-helper-2)

(disable ilessp-itimes-list-eval$-list-delete-prime)

(disable ilessp-0-itimes)

(disable ilessp-itimes-0)
```

```
(disable listp-cdr-factors-implies-integerp)


;; We presumably have better meta-lemmas now, but if we want we
;; can disable those (i.e., correctness-of-cancel-itimes-factors,
;; correctness-of-cancel-itimes-ilessp-factors,
;; correctness-of-cancel-factors-0, and
;; correctness-of-cancel-factors-ilessp-0) and enable the two
;; mentioned below:


(disable correctness-of-cancel-itimes)

(disable correctness-of-cancel-itimes-ilessp)


;; I'll disable some rules now, finally, that I'd previously thought
;; would be OK but now fear because of potential nasty backchaining.


(disable not-integerp-implies-not-equal-iplus)

(disable not-integerp-implies-not-equal-itimes)

(disable subbagp-subsetp)

(disable eval$-list-bagint-0-implies-equal-for-ilessp)

; ---------- Cancel ineg terms from equalities and inequalities ----------


(defn split-out-ineg-terms (x)
  ;; Here x is a list and we return a pair consisting of
  ;; a list of the terms in x not starting with INEG and a list
  ;; of those terms that do (with their INEG headers stripped off).
  (if (listp x)
      (let ((pair (split-out-ineg-terms (cdr x)))
            (a (car x)))
        (if (listp a)
            (if (equal (car a) 'ineg)
                (cons (car pair)
                      (cons (cadr a) (cdr pair)))
              (if (and (equal (car a) 'quote)
                       (negativep (cadr a))
                       (not (equal (negative-guts (cadr a)) 0)))
                  (cons (car pair)
                        (cons (list 'quote (negative-guts (cadr a)))
                              (cdr pair)))
                (cons (cons a (car pair))
                      (cdr pair))))
          (cons (cons a (car pair))
                (cdr pair))))
    (cons nil nil)))
```

```
(defn remove-inegs (x y)
  ;; x and y are term lists that are known to represent integers.
  ;; The idea is to rearrange (equal x y) or (ilessp x y).  Notice
  ;; that the negative terms are put in the front, so that APPEND
  ;; will run fast and do no CONSing in the frequent case that
  ;; there are no negative terms.
  ;;   Returns F, though, if there's no change at all.  I was getting
  ;; into an infinite loop when I built a new term, since there was
  ;; an extra FIX-INT put there.
  (let ((xpair (split-out-ineg-terms x))
        (ypair (split-out-ineg-terms y)))
    (if (or (listp (cdr xpair)) (listp (cdr ypair)))
        (cons (append (cdr ypair) (car xpair))
              (append (cdr xpair) (car ypair)))
      f)))


(defn iplus-or-ineg-term (x)
  (and (listp x)
       (or (equal (car x) (quote ineg))
           (equal (car x) (quote iplus)))))


(defn make-cancel-ineg-terms-equality (x)
  (let ((new-fringes (remove-inegs (iplus-fringe (cadr x))
                                   (iplus-fringe (caddr x)))))
    (if new-fringes
        (if (iplus-or-ineg-term (cadr x))
          (if (iplus-or-ineg-term (caddr x))
              (list (quote equal)
                    (iplus-tree (car new-fringes))
                    (iplus-tree (cdr new-fringes)))
            (list 'if
                  (list 'integerp (caddr x))
                  (list (quote equal)
                        (iplus-tree (car new-fringes))
                        (iplus-tree (cdr new-fringes)))
                  (list 'quote f)))
        ;; otherwise, the first argument is not an iplus or ineg term
        (if (iplus-or-ineg-term (caddr x))
            (list 'if
                  (list 'integerp (cadr x))
                  (list (quote equal)
                        (iplus-tree (car new-fringes))
                        (iplus-tree (cdr new-fringes)))
                  (list 'quote f))
          x))
      x)))
```

```
(defn cancel-ineg-terms-from-equality (x)
  (if (and (listp x)
           (equal (car x) (quote equal)))
      (make-cancel-ineg-terms-equality x)
    x))

;; The following was created from nqthm-macroexpand with arguments
;; and or make-cancel-ineg-terms-equality iplus-or-ineg-term
```

```
(DEFN CANCEL-INEG-TERMS-FROM-EQUALITY-expanded (X)
      (IF (LISTP X)
          (IF (EQUAL (CAR X) 'EQUAL)
              (IF (REMOVE-INEGS (IPLUS-FRINGE (CADR X))
                      (IPLUS-FRINGE (CADDR X)))
                  (COND
                    ((LISTP (CADR X))
                     (CASE (CAR (CAR (CDR X)))
                       (INEG (IF (LISTP (CADDR X))
                                 (CASE (CAR (CAR (CDR (CDR X))))
                                   (INEG
                                    (LIST 'EQUAL
                                     (IPLUS-TREE
                                      (CAR
                                       (REMOVE-INEGS
                                        (IPLUS-FRINGE (CADR X))
                                        (IPLUS-FRINGE (CADDR X)))))
                                     (IPLUS-TREE
                                      (CDR
                                       (REMOVE-INEGS
                                        (IPLUS-FRINGE (CADR X))
                                        (IPLUS-FRINGE (CADDR X)))))))
                                   (IPLUS
                                    (LIST 'EQUAL
                                     (IPLUS-TREE
                                      (CAR
                                       (REMOVE-INEGS
                                        (IPLUS-FRINGE (CADR X))
                                        (IPLUS-FRINGE (CADDR X)))))
                                     (IPLUS-TREE
                                      (CDR
                                       (REMOVE-INEGS
                                        (IPLUS-FRINGE (CADR X))
                                        (IPLUS-FRINGE (CADDR X)))))))
                                   (OTHERWISE
                                    (LIST 'IF
                                     (LIST 'INTEGERP (CADDR X))
                                     (LIST 'EQUAL
                                      (IPLUS-TREE
                                       (CAR
                                        (REMOVE-INEGS
                                         (IPLUS-FRINGE (CADR X))
                                         (IPLUS-FRINGE (CADDR X)))))
                                      (IPLUS-TREE
                                       (CDR
                                        (REMOVE-INEGS
                                         (IPLUS-FRINGE (CADR X))
                                         (IPLUS-FRINGE (CADDR X))))))
                                     (LIST 'QUOTE F))))
                                 (LIST 'IF (LIST 'INTEGERP (CADDR X))
                                       (LIST 'EQUAL
                                        (IPLUS-TREE
                                         (CAR
                                          (REMOVE-INEGS
```

```
                    (IPLUS-FRINGE (CADR X))
                    (IPLUS-FRINGE (CADDR X)))))
                 (IPLUS-TREE
                  (CDR
                   (REMOVE-INEGS
                    (IPLUS-FRINGE (CADR X))
                    (IPLUS-FRINGE (CADDR X))))))
                (LIST 'QUOTE F))))
(IPLUS (IF (LISTP (CADDR X))
           (CASE (CAR (CAR (CDR (CDR X))))
             (INEG
              (LIST 'EQUAL
               (IPLUS-TREE
                (CAR
                 (REMOVE-INEGS
                  (IPLUS-FRINGE (CADR X))
                  (IPLUS-FRINGE (CADDR X)))))
               (IPLUS-TREE
                (CDR
                 (REMOVE-INEGS
                  (IPLUS-FRINGE (CADR X))
                  (IPLUS-FRINGE (CADDR X)))))))
             (IPLUS
              (LIST 'EQUAL
               (IPLUS-TREE
                (CAR
                 (REMOVE-INEGS
                  (IPLUS-FRINGE (CADR X))
                  (IPLUS-FRINGE (CADDR X)))))
               (IPLUS-TREE
                (CDR
                 (REMOVE-INEGS
                  (IPLUS-FRINGE (CADR X))
                  (IPLUS-FRINGE (CADDR X)))))))
             (OTHERWISE
              (LIST 'IF
               (LIST 'INTEGERP (CADDR X))
               (LIST 'EQUAL
                (IPLUS-TREE
                 (CAR
                  (REMOVE-INEGS
                   (IPLUS-FRINGE (CADR X))
                   (IPLUS-FRINGE (CADDR X)))))
                (IPLUS-TREE
                 (CDR
                  (REMOVE-INEGS
                   (IPLUS-FRINGE (CADR X))
                   (IPLUS-FRINGE (CADDR X))))))
               (LIST 'QUOTE F))))
           (LIST 'IF (LIST 'INTEGERP (CADDR X))
                 (LIST 'EQUAL
                  (IPLUS-TREE
                   (CAR
                    (REMOVE-INEGS
                     (IPLUS-FRINGE (CADR X))
```

```
                            (IPLUS-FRINGE (CADDR X)))))
                         (IPLUS-TREE
                          (CDR
                           (REMOVE-INEGS
                            (IPLUS-FRINGE (CADR X))
                            (IPLUS-FRINGE (CADDR X))))))
                        (LIST 'QUOTE F))))
      (OTHERWISE
          (IF (LISTP (CADDR X))
              (CASE (CAR (CAR (CDR (CDR X))))
                (INEG (LIST 'IF
                        (LIST 'INTEGERP (CADR X))
                        (LIST 'EQUAL
                         (IPLUS-TREE
                          (CAR
                           (REMOVE-INEGS
                            (IPLUS-FRINGE (CADR X))
                            (IPLUS-FRINGE (CADDR X)))))
                         (IPLUS-TREE
                          (CDR
                           (REMOVE-INEGS
                            (IPLUS-FRINGE (CADR X))
                            (IPLUS-FRINGE (CADDR X))))))
                        (LIST 'QUOTE F)))
                (IPLUS (LIST 'IF
                         (LIST 'INTEGERP (CADR X))
                         (LIST 'EQUAL
                          (IPLUS-TREE
                           (CAR
                            (REMOVE-INEGS
                             (IPLUS-FRINGE (CADR X))
                             (IPLUS-FRINGE (CADDR X)))))
                          (IPLUS-TREE
                           (CDR
                            (REMOVE-INEGS
                             (IPLUS-FRINGE (CADR X))
                             (IPLUS-FRINGE (CADDR X))))))
                         (LIST 'QUOTE F)))
                (OTHERWISE X))
              X))))
   ((LISTP (CADDR X))
    (CASE (CAR (CAR (CDR (CDR X))))
      (INEG (LIST 'IF (LIST 'INTEGERP (CADR X))
                  (LIST 'EQUAL
                   (IPLUS-TREE
                    (CAR
                     (REMOVE-INEGS
                      (IPLUS-FRINGE (CADR X))
                      (IPLUS-FRINGE (CADDR X)))))
                   (IPLUS-TREE
                    (CDR
                     (REMOVE-INEGS
                      (IPLUS-FRINGE (CADR X))
                      (IPLUS-FRINGE (CADDR X))))))
                  (LIST 'QUOTE F)))
```

```
                     (IPLUS (LIST 'IF (LIST 'INTEGERP (CADR X))
                                  (LIST 'EQUAL
                                   (IPLUS-TREE
                                    (CAR
                                     (REMOVE-INEGS
                                      (IPLUS-FRINGE (CADR X))
                                      (IPLUS-FRINGE (CADDR X)))))
                                   (IPLUS-TREE
                                    (CDR
                                     (REMOVE-INEGS
                                      (IPLUS-FRINGE (CADR X))
                                      (IPLUS-FRINGE (CADDR X))))))
                                  (LIST 'QUOTE F)))
                     (OTHERWISE X)))
                  (T X))
                X)
            X)
        X))


(prove-lemma
 CANCEL-INEG-TERMS-FROM-EQUALITY-CANCEL-INEG-TERMS-FROM-EQUALITY-expanded
 (rewrite)
 (equal (CANCEL-INEG-TERMS-FROM-EQUALITY-expanded x)
        (CANCEL-INEG-TERMS-FROM-EQUALITY x))
 ((disable-theory t)
  (enable-theory ground-zero)
  (enable make-cancel-ineg-terms-equality iplus-or-ineg-term
          CANCEL-INEG-TERMS-FROM-EQUALITY-expanded
          CANCEL-INEG-TERMS-FROM-EQUALITY)))


(disable CANCEL-INEG-TERMS-FROM-EQUALITY-expanded)


(prove-lemma integerp-eval$-iplus-or-ineg-term (rewrite)
  (implies (iplus-or-ineg-term x)
           (integerp (eval$ t x a))))


(disable iplus-or-ineg-term)


(prove-lemma eval$-iplus-list-car-remove-inegs (rewrite)
  (implies (remove-inegs x y)
           (equal (iplus-list (eval$ 'list (car (remove-inegs x y)) a))
                  (iplus (iplus-list
                          (eval$ 'list (car (split-out-ineg-terms x)) a))
                         (iplus-list
                          (eval$ 'list (cdr (split-out-ineg-terms y)) a))))))
```

```
(prove-lemma eval$-iplus-list-cdr-remove-inegs (rewrite)
  (implies (remove-inegs x y)
           (equal (iplus-list (eval$ 'list (cdr (remove-inegs x y)) a))
                  (iplus (iplus-list
                          (eval$ 'list (car (split-out-ineg-terms y)) a))
                         (iplus-list
                          (eval$ 'list (cdr (split-out-ineg-terms x)) a))))))


(prove-lemma minus-ineg (rewrite)
  (implies (and (numberp x)
                (not (equal x 0)))
           (equal (minus x)
                  (ineg x)))
  ((enable-theory integer-defns)))


(prove-lemma iplus-list-eval$-car-split-out-ineg-terms (rewrite)
  (equal (iplus-list (eval$ 'list (car (split-out-ineg-terms x)) a))
         (iplus (iplus-list (eval$ 'list x a))
                (iplus-list (eval$ 'list (cdr (split-out-ineg-terms x)) a))))
  ((induct (split-out-ineg-terms x))
   (enable eval$-quote)))


(disable remove-inegs)


(prove-lemma correctness-of-cancel-ineg-terms-from-equality ((meta equal))
       (equal (eval$ t x a)
              (eval$ t (cancel-ineg-terms-from-equality-expanded x) a))
       ((enable eval$-iplus-tree iplus-list-eval$-fringe eval$-quote)
        (disable iplus-fringe)))


(defn make-cancel-ineg-terms-inequality (x)
  (let ((new-fringes (remove-inegs (iplus-fringe (cadr x))
                                   (iplus-fringe (caddr x)))))
    (if new-fringes
        (list (quote ilessp)
              (iplus-tree (car new-fringes))
              (iplus-tree (cdr new-fringes)))
      x)))
```

```
(defn cancel-ineg-terms-from-inequality (x)
  (if (and (listp x)
           (equal (car x) (quote ilessp)))
      ;; the tests below are for efficiency only
      (if (iplus-or-ineg-term (cadr x))
          (make-cancel-ineg-terms-inequality x)
        (if (iplus-or-ineg-term (caddr x))
            (make-cancel-ineg-terms-inequality x)
          x))
    x))

;; The following was created from nqthm-macroexpand with arguments
;; and or make-cancel-ineg-terms-inequality iplus-or-ineg-term
```

```
(DEFN CANCEL-INEG-TERMS-FROM-INEQUALITY-expanded (X)
      (IF (LISTP X)
          (IF (EQUAL (CAR X) 'ILESSP)
              (COND
                ((LISTP (CADR X))
                 (CASE (CAR (CAR (CDR X)))
                   (INEG (IF (REMOVE-INEGS (IPLUS-FRINGE (CADR X))
                                 (IPLUS-FRINGE (CADDR X)))
                             (LIST 'ILESSP
                                   (IPLUS-TREE
                                    (CAR
                                     (REMOVE-INEGS
                                      (IPLUS-FRINGE (CADR X))
                                      (IPLUS-FRINGE (CADDR X)))))
                                   (IPLUS-TREE
                                    (CDR
                                     (REMOVE-INEGS
                                      (IPLUS-FRINGE (CADR X))
                                      (IPLUS-FRINGE (CADDR X))))))
                             X))
                   (IPLUS (IF (REMOVE-INEGS (IPLUS-FRINGE (CADR X))
                                  (IPLUS-FRINGE (CADDR X)))
                              (LIST 'ILESSP
                                    (IPLUS-TREE
                                     (CAR
                                      (REMOVE-INEGS
                                       (IPLUS-FRINGE (CADR X))
                                       (IPLUS-FRINGE (CADDR X)))))
                                    (IPLUS-TREE
                                     (CDR
                                      (REMOVE-INEGS
                                       (IPLUS-FRINGE (CADR X))
                                       (IPLUS-FRINGE (CADDR X))))))
                              X))
                   (OTHERWISE
                       (IF (LISTP (CADDR X))
                           (CASE (CAR (CAR (CDR (CDR X))))
                             (INEG (IF
                                    (REMOVE-INEGS
                                     (IPLUS-FRINGE (CADR X))
                                     (IPLUS-FRINGE (CADDR X)))
                                    (LIST 'ILESSP
                                     (IPLUS-TREE
                                      (CAR
                                       (REMOVE-INEGS
                                        (IPLUS-FRINGE (CADR X))
                                        (IPLUS-FRINGE (CADDR X)))))
                                     (IPLUS-TREE
                                      (CDR
                                       (REMOVE-INEGS
                                        (IPLUS-FRINGE (CADR X))
                                        (IPLUS-FRINGE (CADDR X))))))
                                    X))
                             (IPLUS (IF
```

```
                      (REMOVE-INEGS
                       (IPLUS-FRINGE (CADR X))
                       (IPLUS-FRINGE (CADDR X)))
                      (LIST 'ILESSP
                       (IPLUS-TREE
                        (CAR
                         (REMOVE-INEGS
                          (IPLUS-FRINGE (CADR X))
                          (IPLUS-FRINGE (CADDR X)))))
                       (IPLUS-TREE
                        (CDR
                         (REMOVE-INEGS
                          (IPLUS-FRINGE (CADR X))
                          (IPLUS-FRINGE (CADDR X))))))
                      X))
              (OTHERWISE X))
             X))))
   ((LISTP (CADDR X))
    (CASE (CAR (CAR (CDR (CDR X))))
      (INEG (IF (REMOVE-INEGS (IPLUS-FRINGE (CADR X))
                    (IPLUS-FRINGE (CADDR X)))
              (LIST 'ILESSP
                    (IPLUS-TREE
                     (CAR
                      (REMOVE-INEGS
                       (IPLUS-FRINGE (CADR X))
                       (IPLUS-FRINGE (CADDR X)))))
                    (IPLUS-TREE
                     (CDR
                      (REMOVE-INEGS
                       (IPLUS-FRINGE (CADR X))
                       (IPLUS-FRINGE (CADDR X))))))
              X))
      (IPLUS (IF (REMOVE-INEGS (IPLUS-FRINGE (CADR X))
                     (IPLUS-FRINGE (CADDR X)))
               (LIST 'ILESSP
                     (IPLUS-TREE
                      (CAR
                       (REMOVE-INEGS
                        (IPLUS-FRINGE (CADR X))
                        (IPLUS-FRINGE (CADDR X)))))
                     (IPLUS-TREE
                      (CDR
                       (REMOVE-INEGS
                        (IPLUS-FRINGE (CADR X))
                        (IPLUS-FRINGE (CADDR X))))))
               X))
      (OTHERWISE X)))
   (T X))
  X)
X))
```

```
(prove-lemma
 CANCEL-INEG-TERMS-FROM-INEQUALITY-CANCEL-INEG-TERMS-FROM-INEQUALITY-expanded
 (rewrite)
 (equal (CANCEL-INEG-TERMS-FROM-INEQUALITY-expanded x)
        (CANCEL-INEG-TERMS-FROM-INEQUALITY x))
 ((disable-theory t)
  (enable-theory ground-zero)
  (enable make-cancel-ineg-terms-inequality iplus-or-ineg-term
          CANCEL-INEG-TERMS-FROM-INEQUALITY-expanded
          CANCEL-INEG-TERMS-FROM-INEQUALITY)))


(disable CANCEL-INEG-TERMS-FROM-INEQUALITY-expanded)


(prove-lemma correctness-of-cancel-ineg-terms-from-inequality ((meta ilessp))
       (equal (eval$ t x a)
              (eval$ t (cancel-ineg-terms-from-inequality-expanded x) a))
       ((enable eval$-iplus-tree iplus-list-eval$-fringe eval$-quote)
        (disable iplus-fringe)))


(disable minus-ineg)

(disable integerp-eval$-iplus-or-ineg-term)

; ---------- Eliminating constants ----------

;; We want to combine in terms like (iplus 3 (iplus x 7)).  Also, when
;; two iplus terms are equated or in-equated, there should only be a
;; natural number summand on at most one side.  Finally, if one adds 1
;; to the right side of a strict inequality, a stronger inequality (in
;; a certain sense) is obtained by removing the 1 and making a non-strict
;; inequality in the other direction.


(prove-lemma plus-iplus (rewrite)
  (implies (and (numberp i) (numberp j))
           (equal (plus i j) (iplus i j)))
  ((enable iplus)))


(prove-lemma iplus-constants (rewrite)
  ;; by now the term presumably has no MINUS terms in it
  (equal (iplus (add1 i) (iplus (add1 j) x))
         (iplus (plus (add1 i) (add1 j)) x))
  ((enable fix-int integerp)
   (disable plus-add1-arg1)))
```

```
(prove-lemma numberp-is-integerp (rewrite)
  (implies (numberp w)
           (integerp w))
  ((enable integerp)))


(prove-lemma difference-idifference (rewrite)
  (implies (and (numberp x)
                (numberp y)
                (leq x y))
           (equal (difference y x)
                  (idifference y x))))


(prove-lemma cancel-constants-equal-lemma nil
  (implies (and (numberp m) (numberp n))
           (equal (equal (iplus m x) (iplus n y))
                  (if (lessp m n)
                      (equal (fix-int x) (iplus (difference n m) y))
                    (equal (iplus (difference m n) x) (fix-int y))))))


(prove-lemma cancel-constants-equal (rewrite)
  (equal (equal (iplus (add1 i) x) (iplus (add1 j) y))
         (if (lessp i j)
             (equal (fix-int x) (iplus (difference j i) y))
           (equal (iplus (difference i j) x) (fix-int y))))
  ((use (cancel-constants-equal-lemma (m (add1 i)) (n (add1 j))))
   (expand (difference (add1 i) (add1 j))
           (difference (add1 j) (add1 i))
           (lessp (add1 i) (add1 j)))
   (disable-theory t)
   (enable-theory ground-zero)))


(prove-lemma ilessp-add1 (rewrite)
  (implies (numberp y)
           (equal (ilessp x (add1 y))
                  (not (ilessp y x))))
  ((enable-theory integer-defns)))


(prove-lemma ilessp-add1-iplus (rewrite)
  (implies (numberp y)
           (equal (ilessp x (iplus (add1 y) z))
                  (not (ilessp (iplus y z) x))))
  ((enable-theory integer-defns)
   (disable plus-iplus difference-idifference)))
```

```
(prove-lemma cancel-constants-ilessp-lemma-1 nil
  (implies (and (numberp m) (numberp n))
           (equal (ilessp (iplus m x) (iplus n y))
                  (if (lessp m n)
                      (ilessp x (iplus (difference n m) y))
                    (ilessp (iplus (difference m n) x) y)))))


(prove-lemma cancel-constants-ilessp-lemma-2 nil
  (implies (and (numberp m) (numberp n))
           (equal (ilessp (iplus m x) (iplus n y))
                  (if (lessp m n)
                      (not (ilessp (iplus (sub1 (difference n m)) y) x))
                    (ilessp (iplus (difference m n) x) y))))
  ((use (cancel-constants-ilessp-lemma-1)
        (ilessp-add1-iplus (y (sub1 (difference n m))) (z y) (x x)))
   (disable ilessp-add1-iplus)))


(prove-lemma cancel-constants-ilessp (rewrite)
  (equal (ilessp (iplus (add1 i) x) (iplus (add1 j) y))
         (if (lessp i j)
             (not (ilessp (iplus (sub1 (difference j i)) y) x))
           (ilessp (iplus (difference i j) x) y)))
  ((use (cancel-constants-ilessp-lemma-2 (m (add1 i)) (n (add1 j))))
   (expand (difference (add1 i) (add1 j))
           (difference (add1 j) (add1 i))
           (lessp (add1 i) (add1 j)))
   (disable-theory t)
   (enable-theory ground-zero)))


(disable plus-iplus)

(disable numberp-is-integerp)

(disable difference-idifference)


; ---------- Final DEFTHEORY event ----------

;; I'll go ahead and include iplus-list and itimes-list and lemmas
;; about them that were developed.

;; I've left out ILESSP-TRICHOTOMY because I'm scared it will slow
;; things down too much.  But it certainly represents useful
;; information.
```

```
(deftheory integers
  (ileq idifference integerp-fix-int
        integerp-iplus integerp-idifference integerp-ineg integerp-iabs
        integerp-itimes fix-int-remover fix-int-fix-int fix-int-iplus
        fix-int-idifference fix-int-ineg fix-int-iabs fix-int-itimes
        ineg-iplus ineg-ineg ineg-fix-int
        ineg-of-non-integerp ineg-0 iplus-left-id iplus-right-id iplus-0-left
        iplus-0-right
        commutativity2-of-iplus commutativity-of-iplus
        associativity-of-iplus iplus-cancellation-1 iplus-cancellation-2
        iplus-ineg1 iplus-ineg2 iplus-fix-int1 iplus-fix-int2
        idifference-fix-int1 idifference-fix-int2
        ;; iplus-fringe lessp-count-listp-cdr iplus-tree-rec iplus-tree
        iplus-list
        ;; eval$-iplus-tree-rec eval$-iplus-tree
        eval$-list-append
        ;; cancel-iplus
        iplus-list-append
        iplus-ineg3 iplus-ineg4
        ;; iplus-list-eval$-fringe
        ;; not-integerp-implies-not-equal-iplus
        ;; <<<disabled because of backchaining>>>
        correctness-of-cancel-iplus
        ilessp-fix-int-1 ilessp-fix-int-2
        ;; make-cancel-iplus-inequality-1 cancel-iplus-ilessp-1
        ;; <<< I omit the following two facts because they're naturals facts,
        ;;    and hence I feel that it's up to naturals to "export"
        ;;    them >>>
        ;; lessp-difference-plus-arg1 lessp-difference-plus-arg1-commuted
        iplus-cancellation-1-for-ilessp iplus-cancellation-2-for-ilessp
        ;; correctness-of-cancel-iplus-ilessp-lemma iplus-tree-no-fix-int
        ;; eval$-ilessp-iplus-tree-no-fix-int
        ;; make-cancel-iplus-inequality-simplifier cancel-iplus-ilessp
        correctness-of-cancel-iplus-ilessp
        ;; itimes-zero1
        itimes-0-left
        ;; itimes-zero2
        itimes-0-right
        itimes-fix-int1 itimes-fix-int2 commutativity-of-itimes
        itimes-distributes-over-iplus-proof itimes-distributes-over-iplus
        commutativity2-of-itimes associativity-of-itimes equal-itimes-0
        equal-itimes-1 equal-itimes-minus-1 itimes-1-arg1
        quotient-remainder-uniqueness
        ;; division-theorem-part1 division-theorem-part2 division-theorem-part3
        division-theorem
        ;; <<< Same comment as in angle braces above >>>
        ;; quotient-difference-lessp-arg2
        ;; iquotient-iremainder-uniqueness
        ;; division-theorem-for-truncate-to-neginf-part1
        ;; division-theorem-for-truncate-to-neginf-part2
        ;; division-theorem-for-truncate-to-neginf-part3
        ;; division-theorem-for-truncate-to-neginf
        ;; idiv-imod-uniqueness
        ;; division-theorem-for-truncate-to-zero-part1
```

```
;; division-theorem-for-truncate-to-zero-part2
;; division-theorem-for-truncate-to-zero-part3
;; division-theorem-for-truncate-to-zero iquo-irem-uniqueness
itimes-ineg-1 itimes-ineg-2 itimes-cancellation-1
itimes-cancellation-2 itimes-cancellation-3 integerp-iquotient
integerp-iremainder integerp-idiv integerp-imod integerp-iquo
integerp-irem iquotient-fix-int1 iquotient-fix-int2
iremainder-fix-int1 iremainder-fix-int2 idiv-fix-int1 idiv-fix-int2
imod-fix-int1 imod-fix-int2 iquo-fix-int1 iquo-fix-int2
irem-fix-int1 irem-fix-int2 fix-int-iquotient fix-int-iremainder
fix-int-idiv fix-int-imod fix-int-iquo fix-int-irem
;; itimes-fringe
;; itimes-tree-rec itimes-tree
itimes-list
;; eval$-itimes-tree-rec
;; eval$-itimes-tree make-cancel-itimes-equality
;; cancel-itimes
itimes-list-append
;; itimes-list-eval$-fringe
;; integerp-eval$-itimes
;; not-integerp-implies-not-equal-itimes
;; <<<disabled because of backchaining>>>
;; itimes-list-eval$-delete itimes-list-bagdiff
;; equal-itimes-list-eval$-list-delete
member-append;; <<< I'll go ahead and export this since it's
;;     so fundamental if one has member around. >>>
;; member-izerop-itimes-fringe correctness-of-cancel-itimes-hack-1
;; eval$-make-cancel-itimes-equality
;; eval$-make-cancel-itimes-equality-1
equal-fix-int
;; eval$-make-cancel-itimes-equality-2
;; eval$-equal-itimes-tree-itimes-fringe-0
;; izerop-eval-of-member-implies-itimes-list-0
subsetp;; <<< May as well have this enabled if it's going to
;;     be imported here. >>>
;; subsetp-implies-itimes-list-eval$-equals-0
;; subbagp-subsetp <<<disabled because of backchaining>>>
;; equal-0-itimes-list-eval$-bagint-1
;; equal-0-itimes-list-eval$-bagint-2
;; correctness-of-cancel-itimes-hack-2
;; correctness-of-cancel-itimes-hack-3-lemma
;; correctness-of-cancel-itimes-hack-3
correctness-of-cancel-itimes
;; itimes-tree-no-fix-int eval$-itimes-tree-no-fix-int-1
;; eval$-itimes-tree-no-fix-int-2 make-cancel-itimes-inequality
;; cancel-itimes-ilessp eval$-make-cancel-itimes-inequality
;; listp-bagint-with-singleton-implies-member itimes-list-eval$-list-0
;; ilessp-itimes-right-positive
;; correctness-of-cancel-itimes-ilessp-hack-1
;; listp-bagint-with-singleton-member
;; <<< Too obscure to be worthwhile >>>
;; correctness-of-cancel-itimes-ilessp-hack-2-lemma
;; correctness-of-cancel-itimes-ilessp-hack-2
;; ilessp-trichotomy
;; <<<Slowed down CORRECTNESS-OF-CANCEL-ITIMES-ILESSP-FACTORS,
```

```
;;                      which might never have even completed.>>>
;; correctness-of-cancel-itimes-ilessp-hack-3-lemma-1
;; correctness-of-cancel-itimes-ilessp-hack-3-lemma-2
;; same-fix-int-implies-not-ilessp
;; correctness-of-cancel-itimes-ilessp-hack-3
;; ilessp-itimes-right-negative
;; correctness-of-cancel-itimes-ilessp-hack-4
correctness-of-cancel-itimes-ilessp ilessp-strict
;; cancel-ineg-aux
;; cancel-ineg
eval$-list-cons eval$-list-nlistp eval$-litatom eval$-quote eval$-other
;; eval$-cancel-ineg-aux-fn eval$-cancel-ineg-aux-is-its-fn
;; iplus-ineg-promote
iplus-x-y-ineg-x
;; correctness-of-cancel-ineg-aux
correctness-of-cancel-ineg integerp-iplus-list
;; eval$-cancel-iplus
eval$-iplus-list-delete eval$-iplus-list-bagdiff
;; iplus-ineg5-lemma-1 iplus-ineg5-lemma-2 iplus-ineg5 iplus-ineg6
;; eval$-iplus plus-ineg7
;; <<<I'm not going to bother disabling functions below.>>>
ITIMES-TREE-INEG ITIMES-FACTORS
ITIMES--1 EQUAL-INEG-INEG ILESSP-INEG-INEG
FIX-INT-EVAL$-ITIMES-TREE-REC   ;may as well leave it enabled
EVAL$-ITIMES-TREE-INEG          ;may as well leave it enabled
INEG-EVAL$-ITIMES-TREE-INEG     ;may as well leave it enabled
IPLUS-EVAL$-ITIMES-TREE-INEG    ;may as well leave it enabled
ITIMES-EVAL$-ITIMES-TREE-INEG   ;may as well leave it enabled
IPLUS-OR-ITIMES-TERM CANCEL-ITIMES-FACTORS
CANCEL-ITIMES-FACTORS-EXPANDED
CANCEL-ITIMES-FACTORS-EXPANDED-CANCEL-ITIMES-FACTORS
;; EQUAL-ITIMES-LIST-EVAL$-LIST-DELETE-NEW-1
;; EQUAL-ITIMES-LIST-EVAL$-LIST-DELETE-NEW-2
;; ITIMES-ITIMES-LIST-EVAL$-LIST-DELETE
;; EQUAL-ITIMES-LIST-EVAL$-LIST-BAGDIFF
MEMBERSHIP-OF-0-IMPLIES-ITIMES-LIST-IS-0 MEMBER-0-EVAL$-LIST
;; ITIMES-LIST-EVAL$-FACTORS-LEMMA
;; ITIMES-LIST-EVAL$-FACTORS-LEMMA-PRIME ITIMES-LIST-EVAL$-FACTORS
;; IPLUS-OR-ITIMES-TERM-INTEGERP-EVAL$
;; EVAL$-LIST-BAGINT-0
;; EVAL$-LIST-BAGINT-0-IMPLIES-EQUAL
CORRECTNESS-OF-CANCEL-ITIMES-FACTORS
CANCEL-ITIMES-ILESSP-FACTORS
BAGINT-SINGLETON
;; <<<A lot of these could be left enabled, in case the metalemmas were
;;    disabled.  But in fact, if one had a reason to disable the
;;    metalemmas then probably one would want most of these
;;    disabled too.>>>
;; IZEROP-ILESSP-0-RELATIONSHIP
;; ILESSP-ITIMES-LIST-EVAL$-LIST-DELETE-HELPER-1
;; ILESSP-ITIMES-LIST-EVAL$-LIST-DELETE-HELPER-2
;; ILESSP-ITIMES-LIST-EVAL$-LIST-DELETE
;; ILESSP-ITIMES-LIST-EVAL$-LIST-DELETE-PRIME-HELPER-1
;; ILESSP-ITIMES-LIST-EVAL$-LIST-DELETE-PRIME-HELPER-2
;; ILESSP-ITIMES-LIST-EVAL$-LIST-DELETE-PRIME
```

```
;; ILESSP-0-ITIMES
;; ILESSP-ITIMES-0
ILESSP-ITIMES-LIST-EVAL$-LIST-BAGDIFF
;; ZERO-ILESSP-IMPLIES-NOT-EQUAL <<<not even a rewrite rule!>>>
ILESSP-ITIMES-LIST-EVAL$-LIST-BAGDIFF-COROLLARY-1
;; ILESSP-ZERO-IMPLIES-NOT-EQUAL <<<not even a rewrite rule!>>>
MEMBER-0-ITIMES-FACTORS-YIELDS-0
MEMBER-0-ITIMES-FACTORS-YIELDS-0-ILESSP-CONSEQUENCE-1
MEMBER-0-ITIMES-FACTORS-YIELDS-0-ILESSP-CONSEQUENCE-2
;; EVAL$-LIST-BAGINT-0-FOR-ILESSP <<<not even a rewrite rule!>>>
;; EVAL$-LIST-BAGINT-0-IMPLIES-EQUAL-FOR-ILESSP-LEMMA
;; <<<not even a rewrite rule!>>>
;; EQUAL-FIX-INT-TO-ILESSP <<<not even a rewrite rule!>>>
;; EVAL$-LIST-BAGINT-0-IMPLIES-EQUAL-FOR-ILESSP
ILESSP-ITIMES-LIST-EVAL$-LIST-BAGDIFF-COROLLARY-2
CORRECTNESS-OF-CANCEL-ITIMES-ILESSP-FACTORS
DISJOIN-EQUALITIES-WITH-0
CANCEL-FACTORS-0 SOME-EVAL$S-TO-0
EVAL$-DISJOIN-EQUALITIES-WITH-0 SOME-EVAL$S-TO-0-APPEND
SOME-EVAL$S-TO-0-ELIMINATOR
;; LISTP-CDR-FACTORS-IMPLIES-INTEGERP
CORRECTNESS-OF-CANCEL-FACTORS-0
CONJOIN-INEQUALITIES-WITH-0
CANCEL-FACTORS-ILESSP-0
;;;;;; and now from the final two metalemmas
split-out-ineg-terms ;; function
;; remove-inegs ;; function, disabled
;; make-cancel-ineg-terms-equality ;; function
;; iplus-or-ineg-term ;; function, disabled
;; cancel-ineg-terms-from-equality ;; function, disabled
;; CANCEL-INEG-TERMS-FROM-EQUALITY-expanded ;; function
;; CANCEL-INEG-TERMS-FROM-EQUALITY-CANCEL-INEG-TERMS-FROM-\
;;       EQUALITY-expanded
;; harmless
;; integerp-eval$-iplus-or-ineg-term
;; should be disabled, since iplus-or-ineg-term is
;; eval$-iplus-list-car-remove-inegs ;; harmless
;; eval$-iplus-list-cdr-remove-inegs ;; harmless
;; minus-ineg ;; definitely should be disabled
;; iplus-list-eval$-car-split-out-ineg-terms ;; harmless
correctness-of-cancel-ineg-terms-from-equality
;; make-cancel-ineg-terms-inequality ;; function
;; cancel-ineg-terms-from-inequality ;; function
;; CANCEL-INEG-TERMS-FROM-INEQUALITY-expanded ;; function, disabled
;; CANCEL-INEG-TERMS-FROM-INEQUALITY-CANCEL-INEG-TERMS-FROM-\
;;       INEQUALITY-expanded
;; harmless
correctness-of-cancel-ineg-terms-from-inequality
;; plus-iplus
iplus-constants
;; numberp-is-integerp
;; difference-idifference
;; cancel-constants-equal-lemma ;; nil lemma
cancel-constants-equal
ilessp-add1
```

```
        ilessp-add1-iplus
        ;; cancel-constants-ilessp-lemma-1 ;; nil lemma
        ;; cancel-constants-ilessp-lemma-2 ;; nil lemma
        cancel-constants-ilessp
        ))

;;;  Added by Warren A. Hunt, Jr.   ---   Thu Sep 12 09:28:07 CDT 1991


(enable eval$)
```

14.19 "math-disable.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.


;;;  This file is to disable everything prior to this point.
;;;  The purpose of this is to make sure that the previous
;;;  events do not interfere with the events that follow.



(disable-back-to ground-zero math-theory)
```

14.20 "intro.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    INTRO.EVENTS
;;;
;;;    Definitions of list manipulation functions, and lemmas about these
;;;    functions.  Also some arithmetic.
;;;
;;;  Some of the lemmas in this file are redundant in the sense that they exist
;;;  in some of the "standard" libraries.  However, when we began the proof
;;;  effort we did not do so in the context of these libraries.  For various
;;;  reasons it was easier to rename our redundant lemmas than completely
;;;  eliminate them.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;    IF*, OR*, AND*, NOT*  -- IF, OR, AND, and NOT, renamed to avoid
;;;    normalization explosion.


(defn if* (a b c) (if a b c))


(disable if*)


(defn or* (a b) (if* a t (if* b t f)))


(disable or*)


(defn and* (a b) (if* a (if* b t f) f))


(disable and*)


(defn not* (a) (if* a f t))


(disable not*)


(deftheory prop*-theory (or* and* not*))
```

```
(prove-lemma simplify-if* (rewrite)
  (and
   (implies
    c
    (equal (if* c a b)
           a))
   (implies
    (not c)
    (equal (if* c a b)
           b)))
  ;;Hint
  ((enable if*)))


(prove-lemma if*-c-x-x (rewrite)
  (equal (if* c x x) x)
  ;;Hint
  ((enable if*)))


(prove-lemma if*-cons (rewrite)
  (equal (if* a (cons b c) (cons d e))
         (cons (if* a b d) (if* a c e)))
  ;;Hint
  ((enable if*)))


(prove-lemma rewrite-not* (rewrite)
  (and
   (implies
    x
    (equal (not* x)
           f))
   (implies
    (not x)
    (equal (not* x)
           t)))
  ;;Hint
  ((enable if* not*)))


(prove-lemma rewrite-and* (rewrite)
  (and (equal (and* f x) f)
       (equal (and* x f) f)
       (implies
        y
        (and (equal (and* y x) (if* x t f))
             (equal (and* x y) (if* x t f)))))
  ;;Hint
  ((enable if* and*)))
```

```
(prove-lemma rewrite-or* (rewrite)
  (and (equal (or* f x) (if* x t f))
       (equal (or* x f) (if* x t f))
       (implies
        y
        (and (equal (or* y x) t)
             (equal (or* x y) t))))
  ;;Hint
  ((enable if* or*)))


(prove-lemma expand-*-connectives (rewrite)
  (and
   (equal (if* c a b) (if c a b))
   (equal (or* a b) (if a t (if b t f)))
   (equal (and* a b) (if a (if b t f) f))
   (equal (not* a) (not a)))
  ;;Hint
  ((enable if* and* or* not*)))


(disable expand-*-connectives)


;;;    CAR,CDR

;;;  This redundant looking lemma helps avoid case splitting in certain cases.


(prove-lemma car-cdr-if-cons (rewrite)
  (and
   (equal (car (if x (cons a b) (cons c d)))
          (if x a c))
   (equal (cdr (if x (cons a b) (cons c d)))
          (if x b d))))

;;;  Essential for extremely esoteric events.


(prove-lemma our-car-cdr-elim (rewrite)
  (equal (equal l (cons (car l) x))
         (and (listp l)
              (equal x (cdr l)))))


(disable our-car-cdr-elim)


;;;   PLUS


(disable plus)
```

```
(prove-lemma plus-0 (rewrite)
  (and
   (implies
    (zerop zero)
    (equal (plus x zero)
           (fix x)))
   (implies
    (zerop zero)
    (equal (plus zero x)
           (fix x))))
  ;;Hint
  ((enable plus)))


(prove-lemma plus-add1 (rewrite)
  (and
   (equal (plus (add1 x) y)
          (add1 (plus x y)))
   (equal (plus x (add1 y))
          (add1 (plus x y)))))


(prove-lemma plus-bottom (rewrite)
  (equal (equal (plus a b) 0)
         (and (zerop a) (zerop b))))


(prove-lemma plus-add1-sub1 (rewrite)
  (and
   (implies
    (not (zerop n))
    (equal (plus (sub1 n) (add1 m))
           (plus n m)))
   (implies
    (not (zerop n))
    (equal (plus (add1 m) (sub1 n))
           (plus m n)))))



;;;    DIFFERENCE


(disable difference)


(prove-lemma difference-x-1 (rewrite)
  (equal (difference x 1)
         (sub1 x))
  ;;Hint
  ((enable difference)))

;#-libraries
```

```
(prove-lemma our-difference-x-x (rewrite)
  (equal (difference x x) 0)
  ;;Hint
  ((enable difference)))

;#-libraries

(prove-lemma our-equal-difference-0 (rewrite)
  (equal (equal (difference x y) 0)
         (leq x y))
  ;;Hint
  ((enable difference)))


(prove-lemma difference-0 (rewrite)
  (implies
   (zerop z)
   (equal (difference x z)
          (fix x)))
  ;;Hint
  ((enable difference)))


(prove-lemma difference-add1-add1 (rewrite)
  (equal (difference (add1 x) (add1 y))
         (difference x y))
  ;;Hint
  ((enable difference)))


(prove-lemma not-lessp-difference (rewrite)
  (equal (lessp x (difference x y))
         f))


(prove-lemma lessp-difference=0 (rewrite)
  (implies
   (leq x y)
   (equal (difference x y)
          0)))



;;;   TIMES


(prove-lemma times-bottom (rewrite)
  (equal (equal (times x y) 0)
         (or (zerop x) (zerop y))))
```

```
(prove-lemma times-1 (rewrite)
  (and (equal (times a 1)
              (fix a))
       (equal (times 1 a)
              (fix a))))


(prove-lemma times-add1-AGAIN (rewrite)
  ;; ***** renamed to avoid conflict with name TIMES-ADD1 in
  ;; naturals library
   (equal (times a (add1 b))
          (plus a (times a b))))



(prove-lemma times-commutes (rewrite)
  (equal (times a b)
         (times b a))
  ;;Hint
  ((enable plus)))



;;;   QUOTIENT


(prove-lemma quotient-lessp (rewrite)
  (implies
   (and (lessp 1 base)
        (not (zerop n)))
   (lessp (quotient n base)
          n)))


(prove-lemma zerop-quotient (rewrite)
  (equal (equal (quotient n m) 0)
         (or (zerop n) (zerop m) (lessp n m))))



;;;  LESSP


(prove-lemma lessp-x-x (rewrite)
  (equal (lessp x x)
         f)
  ;;Hint
  ((enable lessp)))
```

```
(prove-lemma lessp-sub1-x-y-crock (rewrite)
  (implies
   (equal x y)
   (equal (lessp (sub1 x) y)
          (not (zerop x))))
  ;;Hint
  ((enable lessp)))


(prove-lemma lessp-sub1-x-x (rewrite)
  (equal (lessp (sub1 x) x)
         (not (zerop x)))
  ;;Hint
  ((enable lessp)))


(prove-lemma lessp-x-1 (rewrite)
  (equal (lessp x 1)
         (zerop x))
  ;;Hint
  ((enable lessp)))


(prove-lemma leq-lessp-difference (rewrite)
  (implies
   (and (leq x y)
        (not (zerop z))
        (not (zerop y)))
   (equal (lessp (difference x z) y)
          t))
  ;;Hint
  ((enable difference)))


;;;  APPEND


(disable append)


(defn append5 (a b c d e)
  (append a (append b (append c (append d e)))))


(defn append6 (a b c d e g)
  (append a (append b (append c (append d (append e g))))))


(defn append7 (a b c d e g h)
  (append a (append b (append c (append d (append e (append g h)))))))
```

```
(defn append8 (a b c d e g h i)
  (append a
          (append b (append c (append d (append e (append g (append h i))))))))


(prove-lemma append-nlistp (rewrite)
  (implies
   (nlistp a)
   (equal (append a b)
          b))
  ;;Hint
  ((enable append)))


(prove-lemma append-cons (rewrite)
  (equal (append (cons a b) c)
         (cons a (append b c)))
  ;;Hint
  ((enable append)))

;#-libraries

(prove-lemma our-member-append (rewrite)
  (equal (member a (append b c))
         (or (member a b) (member a c)))
  ;;Hint
  ((enable append)))


(prove-lemma associativity-of-append (rewrite)
  (equal (append (append a b) c)
         (append a (append b c)))
  ;;Hint
  ((enable append)))


(prove-lemma equal-append-x-x (rewrite)
  (equal (equal (append a b) (append a c))
         (equal b c))
  ;;Hint
  ((enable append)))



;;;   PROPERP


(defn properp (l)
  (if (listp l)
      (properp (cdr l))
    (equal l nil)))
```

```
(disable properp)


(prove-lemma properp-if (rewrite)
  (implies
   (and (properp a)
        (properp b))
   (properp (if c a b))))


(prove-lemma properp-nlistp (rewrite)
  (implies
   (nlistp l)
   (equal (properp l)
          (equal l nil)))
  ;;Hint
  ((enable properp)))


(prove-lemma properp-cons (rewrite)
  (equal (properp (cons x y))
         (properp y))
  ;;Hint
  ((enable properp)))


(prove-lemma properp-append (rewrite)
  (equal (properp (append a b))
         (properp b))
  ;;Hint
  ((enable properp append)))


(prove-lemma properp-append-nil (rewrite)
  (implies
   (properp a)
   (equal (append a nil)
          a))
  ;;Hint
  ((enable append)))


(prove-lemma properp-if* (rewrite)
  (implies
   (and (properp a)
        (properp b))
   (properp (if* c a b)))
  ;;Hint
  ((enable if*)))


;;;   POSITION
```

```
(defn position (x l)
  (if (nlistp l)
      f
    (if (equal x (car l))
        0
      (if (position x (cdr l))
          (add1 (position x (cdr l)))
        f))))


(disable position)


(prove-lemma member==>position (rewrite)
  (implies
   (member x l)
   (position x l))
  ;;Hint
  ((enable member position)))




;;;   LENGTH


(defn length (l)
  (if (listp l)
      (add1 (length (cdr l)))
    0))


(disable length)


(prove-lemma length-nlistp (rewrite)
  (implies
   (nlistp l)
   (equal (length l)
          0))
  ;;Hint
  ((enable length)))


(prove-lemma length-bottom (rewrite)
  (equal (equal (length x) 0)
         (nlistp x))
  ;;Hint
  ((enable length)))
```

```
(prove-lemma length-1 (rewrite)
  (equal (equal (length a) 1)
         (and (listp a) (nlistp (cdr a))))
  ;;Hint
  ((enable length)))


(prove-lemma length-cons (rewrite)
  (equal (length (cons x y))
         (add1 (length y)))
  ;;Hint
  ((enable length)))


(prove-lemma length-append (rewrite)
  (equal (length (append a b))
         (plus (length a) (length b)))
  ;;Hint
  ((enable length plus append)))


(prove-lemma length-cdr-lemmas (rewrite)
  (and
   (equal (equal (length (cdr x)) (length x))
          (nlistp x))
   (equal (lessp (length (cdr x)) (length x))
          (listp x)))
  ;;Hint
  ((enable lessp length)))


(prove-lemma equal-length-add1 (rewrite)
  (equal (equal (length x) (add1 y))
         (and (listp x)
              (equal (length (cdr x)) (fix y))))
  ;;Hint
  ((enable length)))


(disable equal-length-add1)


(prove-lemma length-cdr (rewrite)
  (equal (equal (length (cdr v))
                (sub1 (length v)))
         t))
```

```
(prove-lemma equal-length-cdr (rewrite)
  (implies
   (equal (length a) (length b))
   (equal (equal (length (cdr a))
                 (length (cdr b)))
          t))
  ;;Hint
  ((enable length)))


(prove-lemma length-if (rewrite)
  (implies
   (equal (length a) (length b))
   (equal (length (if c a b))
          (length a))))


(prove-lemma length-if* (rewrite)
  (implies
   (equal (length a) (length b))
   (equal (length (if* c a b))
          (length a)))
  ;;Hint
  ((enable if*)))



;;;   REVERSE


(defn rev1 (x sponge)
  (if (nlistp x)
      sponge
      (rev1 (cdr x) (cons (car x) sponge))))


(disable rev1)


(defn reverse (x)
  (rev1 x nil))


(disable reverse)


(prove-lemma length-rev1 (rewrite)
  (equal (length (rev1 x sponge))
         (plus (length x)
               (length sponge)))
  ;;Hint
  ((enable rev1)))
```

```
(prove-lemma length-reverse (rewrite)
  (equal (length (reverse x))
         (length x))
  ;;Hint
  ((enable reverse)))



;;;  The standard DELETE deletes only 1 occurrence; DELETE* deletes all.


(defn delete* (x l)
  (if (nlistp l)
      l
    (if (equal x (car l))
        (delete* x (cdr l))
      (cons (car l) (delete* x (cdr l))))))


(disable delete*)



;;;   SUBSET


(defn subset (l1 l2)
  (if (nlistp l1)
      t
    (and (member (car l1) l2)
         (subset (cdr l1) l2))))


(disable subset)


(prove-lemma subset-nlistp (rewrite)
  (implies
   (nlistp x)
   (subset x y))
  ;;Hint
  ((enable subset)))


(prove-lemma subset-cons (rewrite)
  (equal (subset (cons x y) z)
         (and (member x z)
              (subset y z)))
  ;;Hint
  ((enable subset)))
```

```
(prove-lemma subset-x-cons-y-z (rewrite)
  (equal (subset x (cons y z))
         (or (subset x z)
             (and (member y x)
                  (subset (delete* y x) z))))
  ;;Hint
  ((enable subset delete*)))


(prove-lemma subset-append (rewrite)
  (implies
   (or (subset a b) (subset a c))
   (subset a (append b c)))
  ;;Hint
  ((enable subset append)))


(prove-lemma subset-x-x (rewrite)
   (subset x x)
   ;;Hint
   ((enable subset)))




;;;   DISJOINT


(defn disjoint (l1 l2)
  (if (nlistp l1)
      t
    (and (not (member (car l1) l2))
         (disjoint (cdr l1) l2))))


(disable disjoint)


(prove-lemma disjoint-nlistp (rewrite)
  (implies
   (or (nlistp x) (nlistp y))
   (disjoint x y))
  ;;Hint
  ((enable disjoint)))
```

```
(prove-lemma disjoint-cons (rewrite)
  (and
   (equal (disjoint (cons x y) z)
          (and (not (member x z))
               (disjoint y z)))
   (equal (disjoint z (cons x y))
          (and (not (member x z))
               (disjoint z y))))
  ;;Hint
  ((enable disjoint)))


(prove-lemma disjoint-append (rewrite)
  (and
   (equal (disjoint x (append y z))
          (and (disjoint x y)
               (disjoint x z)))
   (equal (disjoint (append y z) x)
          (and (disjoint y x)
               (disjoint z x))))
  ;;Hint
  ((enable disjoint)))



;;;   DUPLICATES?


(defn duplicates? (l)
  (if (nlistp l)
      f
    (or (member (car l) (cdr l))
        (duplicates? (cdr l)))))


(disable duplicates?)


(prove-lemma duplicates?-cons (rewrite)
  (equal (duplicates? (cons x y))
         (or (member x y)
             (duplicates? y)))
  ;;Hint
  ((enable duplicates?)))
```

```
(prove-lemma duplicates-append (rewrite)
  (equal (duplicates? (append a b))
         (or (duplicates? a)
             (duplicates? b)
             (not (disjoint a b))))
  ;;Hint
  ((enable duplicates? disjoint append)))



;;;   FIRSTN


(defn firstn (n l)
  (if (listp l)
      (if (zerop n)
          nil
        (cons (car l) (firstn (sub1 n) (cdr l))))
    nil))


(disable firstn)


(prove-lemma length-firstn (rewrite)
       (equal (length (firstn n l))
              (if (leq n (length l))
                  (fix n)
                  (length l)))
       ((enable firstn length lessp)))


(prove-lemma length-firstn1 (rewrite)
       (implies (leq n (length l))
                (equal (length (firstn n l))
                       (fix n)))
       ((enable length-firstn)))


(prove-lemma length-firstn2 (rewrite)
       (implies (not (leq n (length l)))
                (equal (length (firstn n l))
                       (length l)))
       ((enable length-firstn)))


(prove-lemma car-firstn (rewrite)
  (equal (car (firstn n a))
         (if (and (listp a) (not (zerop n)))
             (car a)
           0))
  ;;Hint
  ((enable firstn)))
```

```
(prove-lemma not-member-firstn (rewrite)
  (implies
   (not (member x y))
   (not (member x (firstn n y))))
  ;;Hint
  ((enable firstn)))


(prove-lemma not-duplicates?-firstn (rewrite)
  (implies
   (not (duplicates? x))
   (not (duplicates? (firstn n x))))
  ;;Hint
  ((enable duplicates? firstn)))


(prove-lemma subset-firstn (rewrite)
  (subset (firstn n l) l)
  ;;Hint
  ((enable subset firstn)))


(prove-lemma firstn-bottom (rewrite)
  (equal (equal (firstn n l) nil)
         (or (nlistp l) (zerop n)))
  ;;Hint
  ((enable firstn)))


(prove-lemma disjoint-firstn (rewrite)
  (implies
   (disjoint x y)
   (disjoint (firstn n x) y))
  ;;Hint
  ((enable disjoint firstn)))


(prove-lemma disjoint-firstn1 (rewrite)
  (implies
   (disjoint y x)
   (disjoint y (firstn n x)))
  ;;Hint
  ((enable disjoint firstn)))


(prove-lemma properp-firstn (rewrite)
  (properp (firstn n l))
  ;;Hint
  ((enable properp firstn)))
```

```
(prove-lemma firstn-append (rewrite)
  (equal (firstn n (append a b))
         (append (firstn n a) (firstn (difference n (length a)) b)))
  ;;Hint
  ((induct (firstn n a))
   (enable firstn append length)))



;;;   RESTN


(defn restn (n l)
  (if (listp l)
      (if (zerop n)
          l
        (restn (sub1 n) (cdr l)))
    l))


(disable restn)


(prove-lemma length-restn (rewrite)
  (equal (length (restn n l))
         (difference (length l) n))
  ((enable restn length difference)))


(prove-lemma not-duplicates?-restn (rewrite)
  (implies
   (not (duplicates? x))
   (not (duplicates? (restn n x))))
  ;;Hint
  ((enable duplicates? restn)))


(prove-lemma not-member-restn (rewrite)
  (implies
   (not (member x y))
   (not (member x (restn n y))))
  ;;Hint
  ((enable restn)))


(prove-lemma subset-restn (rewrite)
  (subset (restn n l) l)
  ;;Hint
  ((enable subset restn)))
```

```
(prove-lemma disjoint-restn (rewrite)
  (implies
   (disjoint x y)
   (disjoint (restn n x) y))
  ;;Hint
  ((enable disjoint restn)))


(prove-lemma disjoint-restn1 (rewrite)
  (implies
   (disjoint y x)
   (disjoint y (restn n x)))
  ;;Hint
  ((enable disjoint restn)))


(prove-lemma properp-restn (rewrite)
  (equal (properp (restn n l))
         (properp l))
  ;;Hint
  ((enable properp restn)))


(prove-lemma restn-append  (rewrite)
  (equal (restn n (append a b))
         (append (restn n a) (restn (difference n (length a)) b)))
  ;;Hint
  ((induct (restn n a))
   (enable restn append)))


(prove-lemma too-many-restns (rewrite)
  (implies
   (and (properp l)
        (leq (length l) n))
   (equal (restn n l)
          nil))
  ;;Hint
  ((induct (restn n l))
   (enable restn properp)))


(prove-lemma cdr-restn (rewrite)
  (implies
   (lessp n (length v))
   (equal (cdr (restn n v))
          (restn (add1 n) v)))
  ;;Hint
  ((enable restn length)))
```

```
(disable cdr-restn)


;;;   FIRSTN/RESTN


(prove-lemma disjoint-firstn-restn-lemmas (rewrite)
  (and
   (implies
    (disjoint x y)
    (disjoint (firstn n x) (firstn m y)))
   (implies
    (disjoint x y)
    (disjoint (firstn n x) (restn m y)))
   (implies
    (disjoint x y)
    (disjoint (restn n x) (firstn m y)))
   (implies
    (disjoint x y)
    (disjoint (restn n x) (restn m y))))
  ;;Hint
  ((enable disjoint firstn restn)))


(prove-lemma no-duplicates-disjoint-firstn-restn (rewrite)
  (and
   (implies
    (not (duplicates? x))
    (disjoint (firstn n x) (restn n x)))
   (implies
    (not (duplicates? x))
    (disjoint (restn n x) (firstn n x))))
  ;;Hint
  ((enable duplicates? disjoint restn firstn)))


(prove-lemma append-firstn-restn (rewrite)
  (equal (append (firstn n l) (restn n l))
         l)
  ;;Hint
  ((enable append firstn restn)))


;;;   PAIRLIST


(disable pairlist)
```

```
(prove-lemma pairlist-nlistp (rewrite)
  (implies
   (nlistp a)
   (equal (pairlist a b)
          nil))
  ;;Hint
  ((enable pairlist)))


(prove-lemma pairlist-cons (rewrite)
  (equal (pairlist (cons x y) z)
         (cons (cons x (car z))
               (pairlist y (cdr z))))
  ;;Hint
  ((enable pairlist)))


(prove-lemma pairlist-append (rewrite)
  (implies
   (equal (length a) (length c))
   (equal (pairlist (append a b) (append c d))
          (append (pairlist a c)
                  (pairlist b d))))
  ;;Hint
  ((enable pairlist append length)))

;;;  Lifted from DUAL-EVAL-BEHAVIORAL.EVENTS


(prove-lemma properp-pairlist (rewrite)
  (properp (pairlist x y))
  ((enable pairlist)))

;;;  This rather redundant looking lemma is needed because there is no way to
;;;  "normalize" an argument of a function without the possibility of infinite
;;;  rewriting  (Except by meta-lemma).   This lemma is "used" by
;;;  EQUAL-COLLECT-EVAL-NAME-PROMOTE-ALISTS.


(prove-lemma pairlists-are-equal-when-their-2nd-lists-are-nlistp (rewrite)
  (implies
   (and (nlistp l1)
        (nlistp l2))
   (equal (equal (pairlist a l1) (pairlist a l2))
          t))
  ;;Hint
  ((enable pairlist)))



;;;  NTH

;;;  NTH - The Nth element of a list.
```

```
(defn nth (n list)
  (if (zerop n)
      (car list)
    (nth (sub1 n) (cdr list))))


(disable nth)

;;; At one point, this lemma seemed to be more trouble than it was worth, but
;;; that may have been due to the poor way that NTH had been defined.  We'll
;;; keep it off, just in case.


(prove-lemma open-nth (rewrite)
  (and
   (implies
    (zerop n)
    (equal (nth n list) (car list)))
   (implies
    (not (zerop n))
    (equal (nth n list)
           (nth (sub1 n) (cdr list)))))
  ;;Hint
  ((enable nth)))


(disable open-nth)

;;;  This funny lemma is used in conjunction with OPEN-NTH, to keep OPEN-NTH
;;;  from being applied through IF's.


(prove-lemma nth-if (rewrite)
  (equal (nth n (if a b c))
         (if a (nth n b) (nth n c))))


(disable nth-if)


(prove-lemma nth-restn (rewrite)
  (equal (nth n v)
         (car (restn n v)))
  ;;Hint
  ((enable length nth restn)))


(disable nth-restn)
```

```
(prove-lemma nth-append (rewrite)
  (implies
   (lessp n (length a))
   (equal (nth n (append a b))
          (nth n a)))
  ;;Hint
  ((induct (nth n a))
   (enable nth append)))


(prove-lemma nth-append-too (rewrite)
  (implies
   (leq (length a) n)
   (equal (nth n (append a b))
          (nth (difference n (length a)) b)))
  ;;Hint
  ((induct (nth n a))
   (enable nth append)))



;;;   NTHCDR


(defn nthcdr (n l)
  (if (zerop n)
      l
    (nthcdr (sub1 n) (cdr l))))


(disable nthcdr)


(prove-lemma open-nthcdr (rewrite)
  (and
   (implies
    (zerop n)
    (equal (nthcdr n l)
           l))
   (implies
    (not (zerop n))
    (equal (nthcdr n l)
           (nthcdr (sub1 n) (cdr l)))))
  ;;Hint
  ((enable nthcdr)))


(prove-lemma properp-as-null-nthcdr (rewrite)
  (equal (properp l)
         (equal (nthcdr (length l) l) nil))
  ;;Hint
  ((enable properp length nthcdr)))
```

```
(disable properp-as-null-nthcdr)

;;;  A dangerous lemma if NTHCDR is enabled.


(prove-lemma cdr-nthcdr (rewrite)
  (equal (cdr (nthcdr n l))
         (nthcdr (add1 n) l))
  ;;Hint
  ((enable nthcdr)))


(disable cdr-nthcdr)


(prove-lemma listp-nthcdr (rewrite)
  (equal (listp (nthcdr n l))
         (lessp n (length l)))
  ;;Hint
  ((enable nthcdr length)))



;;;  SUBRANGE


(defn subrange (l n m)
  (if (lessp m n)
      nil
    (if (zerop n)
        (if (zerop m)
            (list (car l))
          (cons (car l) (subrange (cdr l) 0 (sub1 m))))
      (subrange (cdr l) (sub1 n) (sub1 m)))))


(disable subrange)


(prove-lemma properp-subrange (rewrite)
  (properp (subrange v n m))
  ;;Hint
  ((enable subrange)))

;;;  We only use this lemma in special cases.
```

```
(prove-lemma subrange-cons (rewrite)
  (equal (subrange (cons car cdr) n m)
         (cond ((lessp m n) nil)
               ((zerop n)
                (if (zerop m)
                    (list car)
                  (cons car
                        (subrange cdr 0 (sub1 m)))))
               (t (subrange cdr
                            (sub1 n)
                            (sub1 m)))))
  ;;hint
  ((enable subrange)))


(disable subrange-cons)


(prove-lemma length-subrange (rewrite)
  (equal (length (subrange l n m))
         (if (lessp m n)
             0
           (add1 (difference m n))))
  ;;Hint
  ((enable subrange)))


(prove-lemma subrange-append-right (rewrite)
  (implies
   (and (not (lessp m n))
        (leq (length a) n))
   (equal (subrange (append a b) n m)
          (subrange b (difference n (length a)) (difference m (length a)))))
  ;;Hint
  ((induct (subrange a n m))
   (enable subrange append length)
   (expand (difference n (length a))
           (difference m (length a)))))


(prove-lemma subrange-append-left (rewrite)
  (implies
   (and (not (lessp m n))
        (lessp n (length a))
        (lessp m (length a)))
   (equal (subrange (append a b) n m)
          (subrange a n m)))
  ;;Hint
  ((induct (subrange a n m))
   (enable subrange append length)))
```

```
(prove-lemma subrange-0 (rewrite)
  (implies
   (and (equal m (sub1 (length a)))
        (properp a)
        (not (zerop (length a))))
   (equal (subrange a 0 m)
          a))
  ;;Hint
  ((enable subrange properp)))


(prove-lemma listp-subrange (rewrite)
  (equal (listp (subrange l n m))
         (not (lessp m n)))
  ;;Hint
  ((enable subrange)))


(prove-lemma subset-subrange (rewrite)
  (implies
   (lessp n (length v))
   (subset (subrange v m n) v))
  ;;Hint
  ((enable length subset subrange)
   (expand (subrange v m 0))))


(prove-lemma not-member-subrange (rewrite)
  (implies
   (and (lessp n (length v))
        (not (member x v)))
   (not (member x (subrange v m n))))
  ;;Hint
  ((enable length member subrange)))


(prove-lemma disjoint-subrange (rewrite)
  (implies
   (and (disjoint x y)
        (lessp m (length y)))
   (and (disjoint x (subrange y n m))
        (disjoint (subrange y n m) x)))
  ;;Hint
  ((enable disjoint subrange)))

;;;  A rarely used lemma
```

```
(prove-lemma open-subrange (rewrite)
  (and
   (implies
    (lessp m n)
    (equal (subrange l n m)
           nil))
   (implies
    (and (not (lessp m n))
         (zerop n)
         (zerop m))
    (equal (subrange l n m)
           (list (car l))))
   (implies
    (and (not (lessp m n))
         (zerop n)
         (not (zerop m)))
    (equal (subrange l n m)
           (cons (car l)
                 (subrange (cdr l) 0 (sub1 m)))))
   (implies
    (and (not (lessp m n))
         (not (zerop n)))
    (equal (subrange l n m)
           (subrange (cdr l)
                     (sub1 n)
                     (sub1 m)))))
  ;;hint
  ((enable subrange)))


(disable open-subrange)



;;;  UPDATE-NTH


(defn update-nth (n lst value)
  (if (listp lst)
      (if (zerop n)
          (cons value (cdr lst))
          (cons (car lst)
                (update-nth (sub1 n) (cdr lst) value)))
      lst))


(disable update-nth)



;;;  MAKE-LIST
```

```
(defn make-list (n value)
  (if (zerop n)
      nil
    (cons value (make-list (sub1 n) value))))


(disable make-list)


(prove-lemma open-make-list (rewrite)
  (and
  (implies
   (zerop n)
   (equal (make-list n x)
          nil))
  (implies
   (not (zerop n))
   (equal (make-list n x)
          (cons x (make-list (sub1 n) x)))))
  ;;Hint
  ((enable make-list)))


(disable open-make-list)


(prove-lemma length-make-list (rewrite)
  (equal (length (make-list n value))
         (fix n))
  ;;Hint
  ((enable make-list)))


(prove-lemma make-list-append (rewrite)
  (equal (append (make-list n value)
                 (make-list m value))
         (make-list (plus n m) value))
  ;;Hint
  ((enable make-list)))


(prove-lemma properp-make-list (rewrite)
  (properp (make-list n value))
  ;;Hint
  ((enable make-list)))
```

```
(prove-lemma firstn-make-list (rewrite)
  (implies
   (leq n m)
   (equal (firstn n (make-list m v))
          (make-list n v)))
  ;;Hint
  ((enable firstn make-list)))



;;;  TREE-SIZE


(defn tree-size (tree)
  (if (nlistp tree)
      1
    (plus (tree-size (car tree))
          (tree-size (cdr tree)))))


(disable tree-size)


(prove-lemma tree-size-nlistp (rewrite)
  (implies
   (nlistp tree)
   (equal (tree-size tree)
          1))
  ;;Hint
  ((enable tree-size)))


(prove-lemma not-equal-tree-size-tree-0 (rewrite)
  (not (equal (tree-size tree) 0))
  ;;Hint
  ((enable tree-size)))


(prove-lemma tree-size-1-crock (rewrite)
  (not (equal 1 (tree-size (cons a b))))
  ;;Hint
  ((enable tree-size)))


(prove-lemma a-helpful-lemma-for-tree-inductions (rewrite)
  (implies
   (equal (length a) (tree-size tree))
   (equal (lessp (length a) (tree-size (car tree)))
          f))
  ;;Hint
  ((enable tree-size lessp)))
```

```
(prove-lemma tree-size-lemmas (rewrite)
  (and
   (implies
    (and (listp tree)
         (equal size (tree-size tree)))
    (equal (difference size (tree-size (car tree)))
           (tree-size (cdr tree))))
   (implies
    (and (listp tree)
         (equal size (tree-size tree)))
    (equal (difference size (tree-size (cdr tree)))
           (tree-size (car tree)))))
  ;;Hint
  ((expand (tree-size tree))))


(prove-lemma make-list-append-tree-crock (rewrite)
  (implies
   (listp tree)
   (equal (make-list (plus (tree-size (car tree))
                           (tree-size (cdr tree)))
                     value)
          (make-list (tree-size tree) value)))
   ;;Hint
   ((enable tree-size)))


(disable make-list-append-tree-crock)

;;;  TFIRSTN


(defn tfirstn (list tree)
  (firstn (tree-size (car tree)) list))

;;;  TRESTN


(defn trestn (list tree)
  (restn (tree-size (car tree)) list))



;;;   TREE-HEIGHT


(defn tree-height (tree)
  (if (nlistp tree)
      1
    (add1 (max (tree-height (car tree))
               (tree-height (cdr tree))))))
```

```
(disable tree-height)




;;;   MAKE-TREE n  -- Makes a tree of size N.


(defn make-tree (n)
  (if (zerop n)
      0
    (if (equal n 1)
        0
      (cons (make-tree (quotient n 2))
            (make-tree (difference n (quotient n 2)))))))


(disable make-tree)


(prove-lemma tree-size-make-tree (rewrite)
  (implies
   (not (zerop n))
   (equal (tree-size (make-tree n))
          n))
  ;;Hint
  ((enable tree-size make-tree)))


(prove-lemma listp-make-tree (rewrite)
  (equal (listp (make-tree n))
         (geq n 2))
  ;;Hint
  ((disable difference-0)
   (enable make-tree)))
```

14.21 "list-rewrites.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    LIST-REWRITES.EVENTS
;;;
;;;    For some events, the NQTHM elimination heuristics are too slow or
;;;    limited to do the job, so we need these helper lemmas.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(prove-lemma list-rewrite-4 (rewrite)
  (implies
   (and (properp a)
        (equal (length a) 4))
   (equal (list (car a) (cadr a) (caddr a) (cadddr a))
          a))
  ;;Hint
  ((enable equal-length-add1)))


(disable list-rewrite-4)

;;;    Not strictly necessary, but sometimes I get bored waiting for the prover
;;;    to get around to doing an elim.


(prove-lemma list-elim-4 (rewrite)
  (equal (equal l (list a b c d))
         (and (equal (car l) a)
              (equal (cadr l) b)
              (equal (caddr l) c)
              (equal (cadddr l) d)
              (equal (cddddr l) nil))))


(disable list-elim-4)

;;;   LIST-AS-COLLECTED-NTH
;;;
;;;   What can I say about this bit of proof hackery?  Along with OPEN-NTH,
;;;   PROPERP-AS-NULL-NTHCDR, and OUR-CAR-CDR-ELIM, a quick and dirty way
;;;   to rewrite PROPERP lists as (LIST (CAR L) (CADR L) .... (CADD....DR L)).
;;;   Useful, since with long lists you may run out of ELIM variables and
;;;   experience the dreaded SET-DIFF-N crash.
```

```
(defn list-as-collected-nth (l length n)
  (if (zerop length)
      nil
    (cons (nth n l)
          (list-as-collected-nth l (sub1 length) (add1 n)))))

(disable list-as-collected-nth)

(prove-lemma open-list-as-collected-nth (rewrite)
  (and
   (implies
    (zerop length)
    (equal (list-as-collected-nth l length n)
           nil))
   (implies
    (not (zerop length))
    (equal (list-as-collected-nth l length n)
           (cons (nth n l)
                 (list-as-collected-nth l (sub1 length) (add1 n))))))
  ;;Hint
  ((enable list-as-collected-nth)))

(prove-lemma equal-length-4-as-collected-nth ()
  (implies
   (and (equal (length l) 4)
        (properp l))
   (equal l (list-as-collected-nth l 4 0)))
  ;;Hint
  ((enable open-nth properp-as-null-nthcdr our-car-cdr-elim)
   (disable car-cdr-elim)))

(prove-lemma equal-length-32-as-collected-nth ()
  (implies
   (and (equal (length l) 32)
        (properp l))
   (equal l (list-as-collected-nth l 32 0)))
  ;;Hint
  ((enable open-nth properp-as-null-nthcdr our-car-cdr-elim)
   (disable car-cdr-elim)))
```

14.22 "indices.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    INDICES.EVENTS -- Index name generators, and lemmas.
;;;
;;;  The shell INDEX is used for indexed names in our DUAL-EVAL netlists, where
;;;  we expect the I-NAME of the INDEX to be a LITATOM, and the I-NUM to be the
;;;  index (a number).  We chose to use a paired name, rather than a naming
;;;  convention (e.g., G_0, G_1 ...) to represent indexed names because it is
;;;  much easier to reason about paired names.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(add-shell index nil indexp
  ((i-name (one-of numberp litatom) zero)
   (i-num  (one-of numberp) zero)))


(defn lnfix (x)
  (if (or (numberp x) (litatom x))
      x
    0))



;;;    INDICES -- A list of N indexed names.


(defn indices (name from n)
  (if (zerop n)
      nil
    (cons (index name from)
          (indices name (add1 from) (sub1 n)))))


(disable indices)


(prove-lemma indices-zerop (rewrite)
  (implies
   (zerop n)
   (equal (indices name from n)
          nil))
  ;;Hint
  ((enable indices)))
```

```
(prove-lemma open-indices (rewrite)
  (implies
   (not (zerop n))
   (equal (indices name from n)
          (cons (index name from)
                (indices name (add1 from) (sub1 n)))))
  ;;Hint
  ((enable indices)))

;;;   INDICES-AS-APPEND is an alternate way to look at the structure of the
;;;   indices.


(prove-lemma indices-as-append (rewrite)
  (implies
   (not (zerop n))
   (equal (indices name from n)
          (append (indices name from (sub1 n))
                  (list (index name (plus (sub1 n) from))))))
  ;;Hint
  ((enable indices)))


(disable indices-as-append)


(prove-lemma length-indices (rewrite)
  (equal (length (indices name from n))
         (fix n))
  ;;Hint
  ((enable length indices)))


(prove-lemma listp-indices (rewrite)
  (equal (listp (indices name from n))
         (not (zerop n)))
  ;;Hint
  ((enable indices)))


(prove-lemma properp-indices (rewrite)
  (properp (indices name from n))
  ;;Hint
  ((enable properp indices)))
```

```
(prove-lemma member-indices (rewrite)
  (equal (member x (indices name from n))
         (and (not (zerop n))
              (indexp x)
              (equal (i-name x) (lnfix name))
              (leq from (i-num x))
              (lessp (i-num x) (plus from n))))
  ;;Hint
  ((enable member indices lessp)))


(prove-lemma disjoint-indices-different-names (rewrite)
  (implies
   (not (equal (lnfix name1) (lnfix name2)))
   (disjoint (indices name1 from1 n1) (indices name2 from2 n2)))
  ;;Hint
  ((enable disjoint indices)))


(prove-lemma no-duplicates-in-indices (rewrite)
  (not (duplicates? (indices name from n)))
  ;;Hint
  ((enable duplicates? indices)))

;;;  For some reason, OPEN-INDICES interferes with these proofs.


(prove-lemma position-name-indices (rewrite)
  (implies
   (member index (indices name from n))
   (equal (position index (indices name from n))
          (difference (i-num index) from)))
  ;;Hint
  ((induct (indices name from n))
   (enable position indices)
   (disable open-indices)))


(prove-lemma nth-indices (rewrite)
  (implies
   (lessp n m)
   (equal (nth n (indices name from m))
          (index name (plus n from))))
  ;;Hint
  ((enable nth indices)))


(disable nth-indices)
```

14.23 "hard-specs.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    HARD-SPECS.EVENTS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  X and Z


(add-shell x () xp ())

(add-shell z () zp ())

;;;  BOOLP etc.


(defn boolp (x)
  (or (equal x t) (equal x f)))


(disable boolp)


(prove-lemma prove-boolp (rewrite)
  (implies
   (or (truep x) (falsep x))
   (boolp x))
  ;;Hint
  ((enable boolp)))


(prove-lemma boolp-lemmas (rewrite)
  (and
   (boolp t)
   (boolp f)
   (implies
    (boolp b)
    (equal (equal b t) b))
   (implies
    (boolp b)
    (equal (equal b f) (not b))))
  ;;Hint
  ((enable boolp)))
```

```
(prove-lemma boolp-implies-not-equal-z (rewrite)
  (implies
   (boolp x)
   (not (equal x (z)))))


(disable boolp-implies-not-equal-z)


(defn boolfix (x)
  (if x t f))


(defn threep (x)
  (or (equal x t) (equal x f) (equal x (x))))


(defn threefix (x)
  (if (boolp x)
      x
    (x)))


(prove-lemma threefix-idempotent (rewrite)
  (equal (threefix (threefix x))
         (threefix x))
  ;;Hint
  ((enable threefix)))


(prove-lemma threefix=x (rewrite)
  (equal (equal (x) (threefix x))
         (not (boolp x))))


(disable threefix=x)


(defn fourp (x)
  (or (equal x t) (equal f x) (equal x (x)) (equal x (z))))


(defn fourfix (x)
  (if (fourp x)
      x
    (x)))



;;;   BVP
```

```
(defn bvp (x)
  (if (nlistp x)
      (equal x nil)
    (and (boolp (car x))
         (bvp (cdr x)))))


(disable bvp)


(prove-lemma bvp-nlistp (rewrite)
  (implies
   (nlistp x)
   (equal (bvp x)
          (equal x nil)))
  ;;Hint
  ((enable bvp)))


(prove-lemma bvp-cons (rewrite)
  (equal (bvp (cons x y))
         (and (boolp x)
              (bvp y)))
  ;;Hint
  ((enable bvp)))


(prove-lemma bvp-firstn (rewrite)
  (implies (bvp x)
           (bvp (firstn n x)))
  ;;Hint
  ((enable bvp firstn)))


(prove-lemma bvp-restn (rewrite)
  (implies (bvp x)
           (bvp (restn n x)))
  ;;Hint
  ((enable bvp restn)))


(prove-lemma bvp-append (rewrite)
  (implies (bvp a)
           (equal (bvp (append a b))
                  (bvp b)))
  ;;Hint
  ((enable bvp append)))
```

```
(prove-lemma bvp-is-properp (rewrite)
  (implies
   (bvp v)
   (properp v))
  ;;Hint
  ((enable bvp properp)))


(prove-lemma bvp-make-list (rewrite)
  (equal (bvp (make-list n v))
         (or (zerop n) (boolp v)))
  ;;Hint
  ((enable bvp make-list)))


(prove-lemma bvp-nthcdr (rewrite)
  (implies
   (and (bvp l)
        (leq n (length l)))
   (bvp (nthcdr n l)))
  ;;Hint
  ((enable bvp nthcdr length)))


(prove-lemma bvp-if (rewrite)
  (implies
   (and (bvp a) (bvp b))
   (bvp (if c a b))))


(prove-lemma bvp-if* (rewrite)
  (implies
   (and (bvp a)
        (bvp b))
   (bvp (if* c a b)))
  ;;Hint
  ((enable if*)))


;;;  BV2P
```

```
(defn bv2p (x y)
  (and (bvp x)
       (bvp y)
       (equal (length x) (length y))))



;;;  BVP-LENGTH
;;;
;;;  BVP-LENGTH is a concept introduced in order to be able to decide
;;;  (BOOLP (CAR (CD...DR X))) if X is a long-enough BVP.  This decision is
;;;  made by continually rewriting the hypothesis (BVP-LENGTH bvp n).


(defn bvp-length (bvp n)
  (and (bvp bvp)
       (leq n (length bvp))))


(disable bvp-length)


(prove-lemma bvp-length-cdr (rewrite)
  (implies
   (bvp-length x (add1 n))
   (bvp-length (cdr x) n))
  ;;Hint
  ((induct (firstn n x))
   (enable bvp-length bvp length)))


(prove-lemma boolfix-car-x=x (rewrite)
  (implies
   (bvp-length x 1)
   (equal (boolfix (car x))
          (car x)))
  ;;Hint
  ((enable boolfix bvp-length)))


(prove-lemma boolp-car-x (rewrite)
  (implies
   (bvp-length x 1)
   (boolp (car x)))
  ;;Hint
  ((enable boolp bvp-length)))
```

```
(prove-lemma bvp-length-restn (rewrite)
  (implies
   (bvp bvp)
   (equal (bvp-length (restn n bvp) m)
          (leq m (difference (length bvp) n))))
  ;;Hint
  ((enable bvp-length)))



;;;  NTH


(prove-lemma show-that-nth=t (rewrite)
  (implies
   (and (nth n a)
        (bvp a)
        (lessp n (length a)))
   (equal (nth n a)
          t))
  ;;Hint
  ((induct (nth n a))
   (enable bvp nth)))


(disable show-that-nth=t)


(prove-lemma boolp-nth (rewrite)
  (implies
   (bvp a)
   (equal (boolp (nth n a))
          (lessp n (length a))))
  ;;Hint
  ((induct (nth n a))
   (enable nth bvp)))


(prove-lemma another-way-to-look-at-boolp-nth (rewrite)
  (implies
   (and (bvp v)
        (lessp n (length v)))
   (equal (equal (if (nth n v) t f)
                 (nth n v))
          t)))



;;;    PRIMITIVE SPECIFICATIONS
;;;
;;;    We use the "b-" functions instead of simply AND, OR, etc. in order to
;;;    gain ENABLE/DISABLE control.
```

```
(defn xor (a b)                (if a (if b f t) (if b t f)))

(defn b-buf (x)                (if x t f))

(defn b-not (x)                (not x))

(defn b-nand  (a b)            (not (and a b)))

(defn b-nand3 (a b c)          (not (and a b c)))

(defn b-nand4 (a b c d)        (not (and a b c d)))

(defn b-nand5 (a b c d e)      (not (and a b c d e)))

(defn b-nand6 (a b c d e g)    (not (and a b c d e g)))

(defn b-nand8 (a b c d e g h i) (not (and a b c d e g h i)))

(defn b-or   (a b)             (or a b))

(defn b-or3  (a b c)           (or a b c))

(defn b-or4  (a b c d)         (or a b c d))

(defn b-xor  (x y)             (if x (if y f t) (if y t f)))

(defn b-xor3 (a b c)           (b-xor (b-xor a b) c))

(defn b-equv (x y)             (if x (if y t f) (if y f t)))

(defn b-equv3 (a b c)          (b-equv a (b-xor b c)))

(defn b-and  (a b)             (and a b))

(defn b-and3 (a b c)           (and a b c))

(defn b-and4 (a b c d)         (and a b c d))

(defn b-nor  (a b)             (not (or a b)))

(defn b-nor3 (a b c)           (not (or a b c)))

(defn b-nor4 (a b c d)         (not (or a b c d)))

(defn b-nor5 (a b c d e)       (not (or a b c d e)))

(defn b-nor6 (a b c d e g)     (not (or a b c d e g)))
```

```
(defn b-nor8 (a b c d e g h i)  (not (or a b c d e g h i)))


(defn b-if (c a b)              (if c (if a t f) (if b t f)))



;;;  Force B-XOR's to open


(prove-lemma open-b-xor (rewrite)
  (equal (b-xor a b)
         (if a (if b f t) (if b t f))))


(prove-lemma open-b-xor3 (rewrite)
  (equal (b-xor3 a b c)
         (b-xor a (b-xor b c))))


(prove-lemma open-b-equv (rewrite)
  (equal (b-equv a b)
         (if a (if b t f) (if b f t))))


(prove-lemma open-b-equv3 (rewrite)
  (equal (b-equv3 a b c)
         (b-equv a (b-xor b c))))

;;;  Some facts for those times when B-AND is disabled.


(prove-lemma b-and-rewrite (rewrite)
  (and
   (not (b-and f x))
   (not (b-and x f))
   (implies
    (and x y)
    (b-and x y))))

;;  Lets us use buffers in modules at will.


(prove-lemma b-buf-x=x (rewrite)
  (implies
   (boolp x)
   (equal (b-buf x) x)))


;;;  A boolean gate theory.
```

```
(deftheory b-gates
  (
   b-buf b-not
   b-nand b-nand3 b-nand4 b-nand5 b-nand6 b-nand8
   b-or b-or3 b-or4
   b-xor b-xor3
   b-equv b-equv3
   b-and b-and3 b-and4
   b-nor b-nor3 b-nor4 b-nor5 b-nor6 b-nor8
   b-if
   open-b-xor open-b-xor3
   open-b-equv open-b-equv3
   b-and-rewrite
   b-buf-x=x))

;;; This lemma allows us to prove that specifications written in
;;; terms of 4-valued gate-level functions (see below) are equivalent to
;;; Boolean gate-level functions when the inputs are constrained to be Boolean,
;;; without opening up the gate-level definitions, which would potentially
;;; result in massive clauses and/or case splitting.


(prove-lemma boolp-b-gates (rewrite)
  (and
   (boolp (b-buf x))
   (boolp (b-not x))
   (boolp (b-nand a b))
   (boolp (b-nand3 a b c))
   (boolp (b-nand4 a b c d))
   (boolp (b-nand5 a b c d e))
   (boolp (b-nand6 a b c d e g))
   (boolp (b-nand8 a b c d e g h i))
   (boolp (b-or a b))
   (boolp (b-or3 a b c))
   (boolp (b-or4 a b c d))
   (boolp (b-xor x y))
   (boolp (b-xor3 a b c))
   (boolp (b-equv x y))
   (boolp (b-equv3 a b c))
   (boolp (b-and a b))
   (boolp (b-and3 a b c))
   (boolp (b-and4 a b c d))
   (boolp (b-nor a b))
   (boolp (b-nor3 a b c))
   (boolp (b-nor4 a b c d))
   (boolp (b-nor5 a b c d e))
   (boolp (b-nor6 a b c d e g))
   (boolp (b-nor8 a b c d e g h i))
   (boolp (b-if c a b)))
  ;;Hint
  ((enable boolp)))
```

```
(disable boolp-b-gates)

;;;  ID -- The "renaming" gate.


(defn id (x) x)

;;;  These "compound gates" correspond to LSI Logic macrocells.


(defn ao2 (a b c d) (b-nor (b-and a b) (b-and c d)))


(defn ao4 (a b c d) (b-nand (b-or a b) (b-or c d)))


(defn ao6 (a b c)   (b-nor (b-and a b) c))


(defn ao7 (a b c)   (b-nand (b-or a b) c))

;;;  Power and ground


(defn vss () f)


(defn vdd () t)



;;;  VECTOR SPECIFICATIONS
;;;
;;;  We now define our basic vector hardware specification functions.


(defn v-buf (x)
  (if (nlistp x)
      nil
    (cons (b-buf (car x))
          (v-buf (cdr x)))))


(disable v-buf)


(defn v-not (x)
  (if (nlistp x)
      nil
    (cons (b-not (car x))
          (v-not (cdr x)))))
```

```
(disable v-not)


(defn v-and (x y)
  (if (nlistp x)
      nil
    (cons (b-and (car x) (car y))
          (v-and (cdr x) (cdr y)))))


(disable v-and)


(defn v-or (x y)
  (if (nlistp x)
      nil
    (cons (b-or (car x) (car y))
          (v-or (cdr x) (cdr y)))))


(disable v-or)


(defn v-xor (x y)
  (if (nlistp x)
      nil
    (cons (b-xor (car x) (car y))
          (v-xor (cdr x) (cdr y)))))


(disable v-xor)


(defn v-shift-right (a si)
  (if (nlistp a)
      nil
    (append (v-buf (cdr a))
            (cons (boolfix si) nil))))


(disable v-shift-right)


(defn v-lsr (a)
  (v-shift-right a f))


(defn v-ror (a si)
  (v-shift-right a si))


(defn v-asr (a)
  (v-shift-right a (nth (sub1 (length a)) a)))
```

```
(defn v-if (c a b)
  (if (nlistp a)
      nil
    (cons (if (if c (car a) (car b)) t f)
          (v-if c (cdr a) (cdr b)))))

(disable v-if)

;;;  Vector functions return bit vectors.

(prove-lemma bvp-v-buf (rewrite)
  (bvp (v-buf a))
  ;;Hint
  ((enable v-buf)))

(prove-lemma bvp-v-not (rewrite)
  (bvp (v-not a))
  ;;Hint
  ((enable v-not)))

(prove-lemma bvp-v-and (rewrite)
  (bvp (v-and a b))
  ;;Hint
  ((enable v-and)))

(prove-lemma bvp-v-or (rewrite)
  (bvp (v-or  a b))
  ;;Hint
  ((enable v-or)))

(prove-lemma bvp-v-xor (rewrite)
  (bvp (v-xor a b))
  ;;Hint
  ((enable v-xor)))

(prove-lemma bvp-v-shift-right (rewrite)
  (bvp (v-shift-right a si))
  ;;Hint
  ((enable v-shift-right)))

(prove-lemma bvp-v-lsr (rewrite)
  (bvp (v-lsr a))
  ;;Hint
  ((enable v-lsr)))
```

```
(prove-lemma bvp-v-asr (rewrite)
  (bvp (v-asr a))
  ;;Hint
  ((enable v-asr)))


(prove-lemma bvp-v-ror (rewrite)
  (bvp (v-ror a c))
  ;;Hint
  ((enable v-ror)))


(prove-lemma bvp-v-if (rewrite)
  (bvp (v-if c a b))
  ;;Hint
  ((enable v-if)))

;;;  Lengths of vector functions


(prove-lemma length-v-buf (rewrite)
  (equal (length (v-buf a))
         (length a))
  ;;Hint
  ((enable length v-buf)))


(prove-lemma length-v-not (rewrite)
  (equal (length (v-not a))
         (length a))
  ;;Hint
  ((enable length v-not)))


(prove-lemma length-v-and (rewrite)
  (equal (length (v-and a b))
         (length a))
  ;;Hint
  ((enable length v-and)))


(prove-lemma length-v-or (rewrite)
  (equal (length (v-or a b))
         (length a))
  ;;Hint
  ((enable length v-or)))


(prove-lemma length-v-xor (rewrite)
  (equal (length (v-xor a b))
         (length a))
  ;;Hint
  ((enable length v-xor)))
```

```
(prove-lemma length-v-shift-right (rewrite)
  (equal (length (v-shift-right a b))
         (length a))
  ;;Hint
  ((enable length v-shift-right)))


(prove-lemma length-v-lsr (rewrite)
  (equal (length (v-lsr a))
         (length a))
  ;;Hint
  ((enable length v-lsr)))


(prove-lemma length-v-asr (rewrite)
  (equal (length (v-asr a))
         (length a))
  ;;Hint
  ((enable length v-asr)))


(prove-lemma length-v-ror (rewrite)
  (equal (length (v-ror a b))
         (length a))
  ;;Hint
  ((enable length v-ror)))


(prove-lemma length-v-if (rewrite)
  (equal (length (v-if c a b))
         (length a))
  ;;Hint
  ((enable length v-if)))



;;;  APPEND lemmas for vector functions.


(prove-lemma v-and-append-help (rewrite)
  (implies (equal (length a) (length b))
           (equal (append (v-and a b)
                          (v-and d e))
                  (v-and (append a d) (append b e))))
  ;;Hint
  ((enable append length v-and)))
```

```
(prove-lemma v-or-append-help (rewrite)
  (implies (equal (length a) (length b))
           (equal (append (v-or a b)
                          (v-or d e))
                  (v-or (append a d) (append b e))))
  ;;Hint
  ((enable append length v-or)))


(prove-lemma v-xor-append-help (rewrite)
  (implies (equal (length a) (length b))
           (equal (append (v-xor a b)
                          (v-xor d e))
                  (v-xor (append a d) (append b e))))
  ;;Hint
  ((enable append length v-xor)))


(prove-lemma v-not-append-help (rewrite)
  (equal (append (v-not a) (v-not b))
         (v-not (append a b)))
  ;;Hint
  ((enable append length v-not)))


(prove-lemma v-buf-append-help (rewrite)
  (equal (append (v-buf a) (v-buf b))
         (v-buf (append a b)))
  ;;Hint
  ((enable append length v-buf)))


(prove-lemma v-if-append-help (rewrite)
  (implies
   (equal (length a) (length b))
   (equal (append (v-if c a b) (v-if c d e))
          (v-if c (append a d) (append b e))))
  ;;Hint
  ((enable append length v-if)))

;;;  A congruence for V-IF.
```

```
(prove-lemma v-if-c-congruence (rewrite)
  (implies
   c
   (equal (equal (v-if c a b) (v-if t a b))
          t))
  ;;Hint
  ((enable v-if)))



;;;   Vector functions with FIRSTN/RESTN


(prove-lemma v-not-firstn (rewrite)
  (equal (v-not (firstn n l))
         (firstn n (v-not l)))
  ;;Hint
  ((enable firstn restn v-not)))


(prove-lemma v-not-restn (rewrite)
  (equal (v-not (restn n l))
         (restn n (v-not l)))
  ;;Hint
  ((enable firstn restn v-not)))


(prove-lemma firstn-v-not (rewrite)
  (equal (firstn n (v-not l))
         (v-not (firstn n l))))


(disable firstn-v-not)


(prove-lemma restn-v-not (rewrite)
  (equal (restn n (v-not l))
         (v-not (restn n l))))


(disable restn-v-not)

;;;  An interesting fact about V-NOT
```

```
(prove-lemma v-not-inverts-all (rewrite)
  (implies
   (lessp n (length bvp))
   (equal (nth n (v-not bvp))
          (b-not (nth n bvp))))
  ;;Hint
  ((enable nth v-not length boolfix show-that-nth=t)))

;;;  Another fascinating fact.


(prove-lemma v-or-make-list-f (rewrite)
  (implies
   (and (bvp a)
        (equal (length a) n))
   (equal (v-or (make-list n f) a)
          a))
  ;;Hint
  ((enable length make-list v-or)))



;;;  V-TO-NAT and NAT-TO-V


(defn v-to-nat (v)
  (if (nlistp v)
      0
    (plus (if (car v) 1 0)
          (times 2 (v-to-nat (cdr v))))))


(disable v-to-nat)


(defn nat-to-v (x n)
  (if (zerop n)
      nil
    (cons (not (zerop (remainder x 2)))
          (nat-to-v (quotient x 2) (sub1 n)))))


(disable nat-to-v)


(prove-lemma firstn-nat-to-v (rewrite)
  (implies
   (leq n m)
   (equal (firstn n (nat-to-v nat m))
          (nat-to-v nat n)))
  ;;Hint
  ((enable firstn nat-to-v)))
```

```
(prove-lemma restn-nat-to-v-0-hack (rewrite)
  (implies
   (leq n m)
   (equal (restn n (nat-to-v 0 m))
          (nat-to-v 0 (difference m n))))
  ;;Hint
  ((enable restn nat-to-v difference)))


(prove-lemma length-nat-to-v (rewrite)
  (equal (length (nat-to-v n length))
         (fix length))
  ;;Hint
  ((enable length nat-to-v)))


(prove-lemma bvp-nat-to-v (rewrite)
  (bvp (nat-to-v n length))
  ;;Hint
  ((enable bvp nat-to-v)))


(prove-lemma car-nat-to-v-0-is-f (rewrite)
  (implies
   (not (zerop n))
   (not (car (nat-to-v 0 n))))
  ;;Hint
  ((enable nat-to-v)))


(prove-lemma any-of-nat-to-v-0-is-f (rewrite)
  (implies
   (lessp n m)
   (not (nth n (nat-to-v 0 m))))
  ;;Hint
  ((enable nth nat-to-v)))



;;;  V-NTH


(defn v-nth (v-n lst)
  (nth (v-to-nat v-n) lst))


(disable v-nth)

;;;  UPDATE-V-NTH


(defn update-v-nth (v-n lst value)
  (update-nth (v-to-nat v-n) lst value))
```

```
(disable update-v-nth)


;;;  V-NZEROP and V-ZEROP


(defn v-nzerop (x)
  (if (nlistp x)
      f
      (or (car x)
          (v-nzerop (cdr x)))))


(disable v-nzerop)


(defn v-zerop (x)
  (not (v-nzerop x)))


(prove-lemma v-nzerop-as-or-crock (rewrite)
  (and
   (implies
    (v-nzerop (firstn n a))
    (v-nzerop a))
   (implies
    (v-nzerop (restn n a))
    (v-nzerop a)))
  ;;Hint
  ((enable v-nzerop firstn restn)))


(prove-lemma not-v-nzerop-as-and-crock (rewrite)
  (implies
   (and (not (v-nzerop (firstn n a)))
        (not (v-nzerop (restn n a))))
   (not (v-nzerop a)))
  ;;Hint
  ((enable v-nzerop firstn restn)))


(prove-lemma not-v-nzerop-v-xor-x-x (rewrite)
  (not (v-nzerop (v-xor x x)))
  ;;Hint
  ((enable v-xor v-nzerop)))
```

```
(prove-lemma v-xor-nzerop=not-equal (rewrite)
  (implies
   (bv2p a b)
   (equal (v-nzerop (v-xor a b))
          (not (equal a b))))
  ;;Hint
  ((enable v-nzerop v-xor length bvp boolp)))


(prove-lemma v-zerop-make-list-f (rewrite)
  (equal (v-zerop (make-list n f))
         t)
  ;;Hint
  ((enable v-zerop v-nzerop make-list)))


(prove-lemma not-v-nzerop-all-f (rewrite)
  (not (v-nzerop (make-list n f)))
  ;;Hint
  ((enable v-nzerop make-list)))



;;;  V-NEGP


(defn v-negp (x)
  (if (nlistp x)
      f
      (if (nlistp (cdr x))
          (car x)
          (v-negp (cdr x)))))


(disable v-negp)


(prove-lemma boolp-v-negp (rewrite)
  (implies
   (bvp v)
   (boolp (v-negp v)))
  ;;Hint
  ((enable boolp bvp v-negp)))


(prove-lemma v-negp-as-nth (rewrite)
  (implies
   (not (equal (length bv) 0))
   (equal (v-negp bv)
          (nth (sub1 (length bv)) bv)))
  ;;Hint
  ((enable v-negp nth)))
```

```
(disable v-negp-as-nth)


;;;  SIGN-EXTEND

(defn sign-extend (v n)
  (if (zerop n)
      nil
    (if (nlistp v)
        (make-list n f)
      (if (nlistp (cdr v))
          (cons (boolfix (car v)) (make-list (sub1 n) (boolfix (car v))))
        (cons (boolfix (car v)) (sign-extend (cdr v) (sub1 n)))))))

(disable sign-extend)

(prove-lemma length-sign-extend (rewrite)
  (equal (length (sign-extend v n)) (fix n))
  ;;Hint
  ((enable sign-extend length)))

(prove-lemma bvp-sign-extend (rewrite)
   (bvp (sign-extend v n))
   ;;Hint
   ((enable bvp sign-extend)))

(prove-lemma sign-extend-as-append (rewrite)
  (implies
   (and (bvp v)
        (leq (length v) n)
        (not (equal (length v) 0)))
   (equal (sign-extend v n)
          (append v (make-list (difference n (length v))
                               (nth (sub1 (length v)) v)))))
  ;;Hint
  ((induct (sign-extend v n))
   (enable append sign-extend make-list nth difference)))

(disable sign-extend-as-append)


;;;  V-ADDER is a recursive definition of a binary adder.
```

```
(defn v-adder (c a b)
  (if (nlistp a)
      (cons (boolfix c) nil)
    (cons (b-xor3 c (car a) (car b))
          (v-adder (b-or (b-and (car a) (car b))
                         (b-or (b-and (car a) c)
                               (b-and (car b) c)))
                   (cdr a)
                   (cdr b)))))
```

```
(disable v-adder)
```

```
(defn v-adder-output (c a b)
  (firstn (length a) (v-adder c a b)))
```

```
(defn v-adder-carry-out (c a b)
  (nth (length a) (v-adder c a b)))
```

```
(defn v-adder-overflowp (c a b)
  (b-and (b-equv (nth (sub1 (length a)) a)
                 (nth (sub1 (length b)) b))
         (b-xor (nth (sub1 (length a)) a)
                (nth (sub1 (length a)) (v-adder-output c a b)))))
```

```
(defn v-subtracter-output (c a b)
  (v-adder-output (b-not c) (v-not a) b))
```

```
(defn v-subtracter-carry-out (c a b)
  (b-not (v-adder-carry-out (b-not c) (v-not a) b)))
```

```
(defn v-subtracter-overflowp (c a b)
  (v-adder-overflowp (b-not c) (v-not a) b))
```

```
(defn v-inc (x)
  (v-adder-output t x (nat-to-v 0 (length x))))
```

```
(disable v-inc)
```

```
(defn v-dec (x)
  (v-subtracter-output t (nat-to-v 0 (length x)) x))
```

```
(disable v-dec)


(prove-lemma length-of-v-adder (rewrite)
  (equal (length (v-adder c a b))
         (add1 (length a)))
  ;;Hint
  ((enable length v-adder)))


(prove-lemma bvp-v-adder (rewrite)
  (bvp (v-adder c a b))
  ;;Hint
  ((enable bvp v-adder)))


(prove-lemma length-of-v-adder-output (rewrite)
  (equal (length (v-adder-output c a b)) (length a)))


(prove-lemma length-of-v-subtracter-output (rewrite)
  (equal (length (v-subtracter-output c a b)) (length a)))


(prove-lemma bvp-length-v-inc-v-dec (rewrite)
  (and
   (bvp (v-inc x))
   (bvp (v-dec x))
   (equal (length (v-inc x)) (length x))
   (equal (length (v-dec x)) (length x)))
  ;;Hint
  ((enable v-inc v-dec)))

;;;  Prove the correctness of the vector buffer, selector, and adder.


(prove-lemma v-buf-works (rewrite)
  (implies (bvp x)
           (equal (v-buf x) x))
  ;;Hint
  ((enable bvp v-buf)))


(prove-lemma v-if-works (rewrite)
  (implies (bv2p x y)
           (equal (v-if c x y)
                  (if c x y)))
  ;;Hint
  ((enable bvp v-if length)))
```

```
(prove-lemma v-adder-works (rewrite)
  (implies (bv2p x y)
           (equal (v-to-nat (v-adder c x y))
                  (plus (if c 1 0)
                        (v-to-nat x)
                        (v-to-nat y))))
  ;;Hint
  ((enable boolfix bvp length v-to-nat v-adder plus)))

;;; V-THREEFIX -- A useful concept for registers.


(defn v-threefix (v)
  (if (nlistp v)
      nil
    (cons (threefix (car v))
          (v-threefix (cdr v)))))


(disable v-threefix)


(prove-lemma open-v-threefix (rewrite)
  (and
   (implies
    (nlistp v)
    (equal (v-threefix v)
           nil))
   (implies
    (listp v)
    (equal (v-threefix v)
           (cons (threefix (car v))
                 (v-threefix (cdr v))))))
  ;;Hint
  ((enable v-threefix)))


(prove-lemma v-threefix-bvp (rewrite)
  (implies
   (bvp v)
   (equal (v-threefix v)
          v))
  ;;Hint
  ((enable bvp v-threefix)))


(prove-lemma properp-v-threefix (rewrite)
  (properp (v-threefix x))
  ;;Hint
  ((enable properp v-threefix)))
```

```
(prove-lemma length-v-threefix (rewrite)
  (equal (length (v-threefix x))
         (length x))
  ;;Hint
  ((enable length v-threefix)))


(prove-lemma append-v-threefix (rewrite)
  (equal (append (v-threefix a)
                 (v-threefix b))
         (v-threefix (append a b)))
  ;;Hint
  ((enable append v-threefix)))


(prove-lemma v-threefix-append (rewrite)
  (equal (v-threefix (append a b))
         (append (v-threefix a) (v-threefix b))))


(disable v-threefix-append)


(prove-lemma v-threefix-idempotence (rewrite)
  (equal (v-threefix (v-threefix x))
         (v-threefix x))
  ;;Hint
  ((enable v-threefix)))


(prove-lemma bvp-v-threefix (rewrite)
  (implies
   (properp v)
   (equal (bvp (v-threefix v))
          (bvp v)))
  ;;Hint
  ((enable properp bvp v-threefix)))


(prove-lemma v-threefix-make-list-x (rewrite)
  (equal (v-threefix (make-list n (x)))
         (make-list n (x)))
  ;;Hint
  ((enable v-threefix make-list)))



;;;    V-FOURFIX
```

```
(defn v-fourfix (v)
  (if (nlistp v)
      nil
    (cons (fourfix (car v))
          (v-fourfix (cdr v)))))


(disable v-fourfix)


(prove-lemma bvp-v-fourfix (rewrite)
  (implies
   (bvp v)
   (equal (v-fourfix v) v))
  ;;Hint
  ((enable v-fourfix bvp)))


(prove-lemma  v-fourfix-make-list (rewrite)
  (implies
   (or (equal x t)
       (equal x f)
       (equal x (x))
       (equal x (z)))
  (equal (v-fourfix (make-list n x))
         (make-list n x)))
  ;;Hint
  ((enable v-fourfix make-list fourfix)))


(prove-lemma v-threefix-v-fourfix (rewrite)
  (equal (v-threefix (v-fourfix v))
         (v-threefix v))
  ;;Hint
  ((enable v-threefix v-fourfix)))



;;;  V-IFF -- A reducing vector IFF.  Vector equivalence.


(defn v-iff (a b)
  (if (nlistp a)
      t
    (and (iff (car a) (car b))
         (v-iff (cdr a) (cdr b)))))


(disable v-iff)
```

```
(prove-lemma v-iff-x-x (rewrite)
  (v-iff x x)
  ;;Hint
  ((enable v-iff)))


(prove-lemma v-iff-rev1 (rewrite)
  (implies
   (and (equal (length a) (length b))
        (equal (length x) (length y)))
   (equal (v-iff (rev1 a x) (rev1 b y))
          (and (v-iff a b)
               (v-iff x y))))
  ;;Hint
  ((enable v-iff rev1)))


(prove-lemma v-iff-reverse (rewrite)
  (implies
   (equal (length a) (length b))
   (equal (v-iff (reverse a) (reverse b))
          (v-iff a b)))
  ;;Hint
  ((enable reverse)))


(prove-lemma v-iff=equal (rewrite)
  (implies
   (bv2p a b)
   (equal (v-iff a b)
          (equal a b)))
  ;;Hint
  ((enable v-iff boolp)))



;;;  Odds and ends...


(prove-lemma bvp-subrange (rewrite)
  (implies
   (and (bvp v)
        (lessp n (length v)))
   (bvp (subrange v m n)))
  ;;Hint
  ((enable subrange bvp length)))
```

```
(prove-lemma boolp-if* (rewrite)
  (implies
   (and (boolp a)
        (boolp b))
   (boolp (if* c a b)))
  ;;Hint
  ((enable if*)))
```

14.24 "value.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    VALUE.EVENTS -- Lemmas about VALUE and COLLECT-VALUE
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;   VALUE -- The ALIST accessor used by DUAL-EVAL.  Like ASSOC, but returns 0
;;;   if the key is not present, rather than F.  The 0 will be interpreted by
;;;   most of our primitives as the unknown value (X).  This makes it easier to
;;;   debug module definitions.


(defn value (name alist)
  (if (nlistp alist)
      0
    (if (listp (car alist))
        (if (equal (caar alist) name)
            (cdar alist)
          (value name (cdr alist)))
      (value name (cdr alist)))))


(disable value)

;;;   COLLECT-VALUE -- Maps VALUE over a list of names.


(defn collect-value (args alist)
  (if (nlistp args)
      nil
    (cons (value (car args) alist)
          (collect-value (cdr args) alist))))


(disable collect-value)

;;;   Proofs about VALUE
```

```
(prove-lemma rewrite-value (rewrite)
  (equal (value a (cons (cons b c) d))
         (if (equal a b)
             c
           (value a d)))
  ;;Hint
  ((enable value)))

;;;  Necessary for some proofs with lots of signals, otherwise stack overflow
;;;  occurs from deeply nested rewriting!  This lemma also speeds up proofs
;;;  significantly if the modules have lots of single output gates.


(prove-lemma rewrite-value-4x (rewrite)
  (equal (value a
                (cons (cons b c)
                      (cons (cons d e)
                            (cons (cons g h)
                                  (cons (cons i j)
                                        k)))))
         (if (equal a b)
             c
           (if (equal a d)
               e
             (if (equal a g)
                 h
               (if (equal a i)
                   j
                 (value a k))))))
  ;;Hint
  ((enable value)))


(prove-lemma value-append-pairlist (rewrite)
  (equal (value a (append (pairlist b c) d))
         (if (member a b)
             (value a (pairlist b c))
           (value a d)))
  ;;Hint
  ((enable value append pairlist)))

;;;  Not sure we always want this one firing

(prove-lemma value-pairlist (rewrite)
  (implies
   (member name names)
   (equal (value name (pairlist names values))
          (car (restn (position name names) values))))
  ;;Hint
  ((enable value pairlist restn position)))


(disable value-pairlist)
```

```
(prove-lemma value-indices-hack (rewrite)
  (implies
   (and (member the-name (indices name n m))
        (leq m (length values)))
   (equal (value the-name (pairlist (indices name n m) values))
          (nth (difference (i-num the-name) n) values)))
  ;;Hint
  ((enable value-pairlist nth-restn)
   (disable open-indices)))

;;;   Proofs about COLLECT-VALUE


(prove-lemma collect-value-nlistp (rewrite)
  (implies
   (nlistp args)
   (equal (collect-value args alist)
          nil))
  ;;Hint
  ((enable collect-value)))


(prove-lemma listp-collect-value (rewrite)
  (equal (listp (collect-value args alist))
         (listp args))
  ;;Hint
  ((enable collect-value)))


(prove-lemma collect-value-append (rewrite)
  (equal (collect-value (append a b) alist)
         (append (collect-value a alist)
                 (collect-value b alist)))
  ;;Hint
  ((enable collect-value append)))


(prove-lemma length-collect-value (rewrite)
  (equal (length (collect-value args alist))
         (length args))
  ;;Hint
  ((enable length collect-value)))


(prove-lemma collect-value-append-pairlist-when-subset (rewrite)
  (implies
   (subset args1 args2)
   (equal (collect-value args1 (append (pairlist args2 answer)
                                       x))
          (collect-value args1 (pairlist args2 answer))))
  ;;Hint
  ((enable subset collect-value append pairlist)))
```

```
(prove-lemma collect-value-append-pairlist-when-disjoint (rewrite)
  (implies
   (disjoint args1 args2)
   (equal (collect-value args1 (append (pairlist args2 a)
                                       b))
          (collect-value args1 b)))
  ;;Hint
  ((enable disjoint collect-value append pairlist)))


(prove-lemma collect-value-disjoint-pairlist (rewrite)
  (implies
   (not (member a args))
   (equal (collect-value args (cons (cons a b) c))
          (collect-value args c)))
  ;;Hint
  ((enable collect-value)))

;;  Avoids the MEMBER check in the above lemma.

(prove-lemma collect-value-litatom-indices-speedup (rewrite)
  (implies
   (litatom a)
   (equal (collect-value (indices name from to) (cons (cons a b) c))
          (collect-value (indices name from to) c)))
  ;;Hint
  ((enable collect-value)))


(prove-lemma equal-collect-value-promote-alists (rewrite)
  (implies
   (equal alist1 alist2)
   (equal (equal (collect-value args alist1)
                 (collect-value args alist2))
          t)))


(prove-lemma collect-value-firstn (rewrite)
  (implies
   (not (duplicates? args))
   (equal (collect-value (firstn n args)
                         (pairlist args answer))
          (collect-value (firstn n args)
                         (pairlist (firstn n args) (firstn n answer)))))
  ;;Hint
  ((enable duplicates? collect-value firstn pairlist)))
```

```
(prove-lemma collect-value-restn (rewrite)
  (implies
   (not (duplicates? args))
   (equal (collect-value (restn n args)
                         (pairlist args answer))
          (collect-value (restn n args)
                         (pairlist (restn n args) (restn n answer)))))
  ;;Hint
  ((enable duplicates? collect-value restn pairlist)))


(prove-lemma collect-value-cons (rewrite)
  (equal (collect-value (cons a b) alist)
         (cons (value a alist)
               (collect-value b alist)))
  ;;Hint
  ((enable collect-value)))


(prove-lemma singleton-collect-value (rewrite)
  (implies
   (equal (length a) 1)
   (equal (collect-value a alist)
          (list (value (car a) alist))))
  ;;Hint
  ((enable collect-value)))


(prove-lemma properp-collect-value (rewrite)
  (properp (collect-value args alist))
  ;;Hint
  ((enable collect-value properp)))


(prove-lemma collect-value-args-pairlist-args (rewrite)
  (implies
   (and (not (duplicates? args))
        (equal (length args) (length list))
        (properp list))
   (equal (collect-value args (pairlist args list))
          list))
  ;;Hint
  ((enable collect-value pairlist)))

;;;  COLLECT-VALUE-SPLITTING-CROCK is used to force a rewrite which goes
;;;  "against the grain" of the normal rewriting direction of
;;;  COLLECT-VALUE-APPEND.
```

```
(prove-lemma collect-value-splitting-crock-helper ()
  (equal (collect-value (append (firstn n l) (restn n l)) alist)
         (append (collect-value (firstn n l) alist)
                 (collect-value (restn n l) alist)))
  ;;Hint
  ((disable append-firstn-restn)))


(prove-lemma collect-value-splitting-crock ()
  (equal (collect-value l alist)
         (append (collect-value (firstn n l) alist)
                 (collect-value (restn n l) alist)))
  ;;Hint
  ((use (collect-value-splitting-crock-helper))))


(prove-lemma collect-value-make-list (rewrite)
  (equal (collect-value (make-list n name) alist)
         (make-list n (value name alist)))
  ;;Hint
  ((enable collect-value make-list)))

;;;  A weird induction scheme for the following lemma.


(defn cdr-cdr-sub1-sub1-induction (l1 n m l2)
  (if (lessp m n)
      t
    (if (zerop n)
        (if (zerop m)
            t
          (cdr-cdr-sub1-sub1-induction (cdr l1) 0 (sub1 m) (cdr l2)))
      (cdr-cdr-sub1-sub1-induction (cdr l1) (sub1 n) (sub1 m) (cdr l2)))))


(prove-lemma collect-value-subrange-args-pairlist-args (rewrite)
  (implies
   (and (not (duplicates? args))
        (equal (length args) (length list))
        (properp list)
        (lessp n (length args)))
   (equal (collect-value (subrange args m n) (pairlist args list))
          (subrange list m n)))
  ;;Hint
  ((enable collect-value pairlist subrange)
   (induct (cdr-cdr-sub1-sub1-induction args m n list))))
```

14.25 "memory.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    MEMORY.EVENTS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;  This file defines a tree-based formalization of memory.  This
;;;  tree-based memory offers advantages over a linear-list formalization.
;;;  Specifically, reading and writing the memory take O(log n) time and
;;;  CONS operations respectively, where n is the number of words in the
;;;  memory.  Also, we are able to "stub-out", or leave unspecified, large
;;;  sections of the memory.
;;;
;;;  Memory is modeled as a CONS tree, where the leaves of the tree are
;;;  instances of one of three special shells: the shell constructor ROM
;;;  tags read-only locations of the memory, while the shell constructor
;;;  RAM tags read-write locations and STUB represents ``unimplemented''
;;;  portions.  Each instance of the memory shells includes a value, which
;;;  is returned when that memory location is read.  RAM cells may be
;;;  overwritten, but writing to a ROM or STUB cell does not change the
;;;  memory.  ROM cells may only appear at the leaves of the tree, wheras
;;;  STUB cells may appear anywhere.  Although our basic definitions do not
;;;  restrict the types of data stored in memory, we assume throughout the
;;;  specification of the FM9001 (and enforce) the restriction
;;;  that only bit-vectors are stored in memory.
;;;
;;;  The bit-vector that specifies the address is used in an obvious way to
;;;  search the memory tree for the addressed location.  Note, however,
;;;  that the address is reversed prior to the search.  This allows for
;;;  more compact storage for sequences of data.  If the address were not
;;;  reversed, then the memory trees would be subject to branching near the
;;;  root of the tree.  With reversed addresses, the branching is localized
;;;  near the leaves. This is an especially important consideration for the
;;;  main memory of the FM9001.  There, each path through the memory tree
;;;  to a leaf cell is constructed from 32 CONS cells.


(add-shell rom () romp
  ((rom-guts (none-of) zero)))


(add-shell ram () ramp
  ((ram-guts (none-of) zero)))
```

```
(add-shell stub () stubp
  ((stub-guts (none-of) zero)))

;;;    MEMORY-PROPERP -- All memory cells are proper lists of length WIDTH.


(defn memory-properp (n width mem)
  (if (stubp mem)
      (and (properp (stub-guts mem))
           (equal (length (stub-guts mem)) width))
    (if (zerop n)
        (cond
         ((ramp mem) (and (properp (ram-guts mem))
                          (equal (length (ram-guts mem)) width)))
         ((romp mem) (and (properp (rom-guts mem))
                          (equal (length (rom-guts mem)) width)))
         (t f))
      (and (listp mem)
           (memory-properp (sub1 n) width (car mem))
           (memory-properp (sub1 n) width (cdr mem))))))


(disable memory-properp)

;;;    MEMORY-OKP -- All memory cells are BVP lists with length WIDTH.


(defn memory-okp (n width mem)
  (if (stubp mem)
      (and (bvp (stub-guts mem))
           (equal (length (stub-guts mem)) width))
    (if (zerop n)
        (cond
         ((ramp mem) (and (bvp (ram-guts mem))
                          (equal (length (ram-guts mem)) width)))
         ((romp mem) (and (bvp (rom-guts mem))
                          (equal (length (rom-guts mem)) width)))
         (t f))
      (and (listp mem)
           (memory-okp (sub1 n) width (car mem))
           (memory-okp (sub1 n) width (cdr mem))))))


(disable memory-okp)

;;;    READ-MEM
```

```
(defn read-mem1 (v-addr mem)
  (if (stubp mem)
      (stub-guts mem)
      (if (nlistp v-addr)
          (cond ((ramp mem) (ram-guts mem))
                ((romp mem) (rom-guts mem))
                (t 0))
          (if (nlistp mem)
              0
              (if (car v-addr)
                  (read-mem1 (cdr v-addr) (cdr mem))
                  (read-mem1 (cdr v-addr) (car mem)))))))


(disable read-mem1)


(defn read-mem (v-addr mem)
  (read-mem1 (reverse v-addr) mem))


(disable read-mem)

;;;   WRITE-MEM


(defn write-mem1 (v-addr mem value)
  (if (stubp mem)
      mem
      (if (nlistp v-addr)
          (cond ((ramp mem) (ram value))
                (t mem))
          (if (nlistp mem)
              mem
              (if (car v-addr)
                  (cons (car mem)
                        (write-mem1 (cdr v-addr) (cdr mem) value))
                  (cons (write-mem1 (cdr v-addr) (car mem) value)
                        (cdr mem)))))))


(disable write-mem1)


(defn write-mem (v-addr mem value)
  (write-mem1 (reverse v-addr) mem value))


(disable write-mem)

;;;   RAMP-MEM  --  A particular address is RAM.
```

```
(defn ramp-mem1 (v-addr mem)
  (if (stubp mem)
      f
      (if (nlistp v-addr)
          (ramp mem)
          (if (nlistp mem)
              f
              (if (car v-addr)
                  (ramp-mem1 (cdr v-addr) (cdr mem))
                  (ramp-mem1 (cdr v-addr) (car mem)))))))


(disable ramp-mem1)


(defn ramp-mem (v-addr mem)
  (ramp-mem1 (reverse v-addr) mem))


(disable ramp-mem)

;;;    ALL-RAMP-MEM  --  The entire memory is RAM.


(defn all-ramp-mem (n mem)
  (if (stubp mem)
      f
    (if (zerop n)
        (ramp mem)
      (if (nlistp mem)
          f
        (and (all-ramp-mem (sub1 n) (car mem))
             (all-ramp-mem (sub1 n) (cdr mem)))))))


(disable all-ramp-mem)

;;;    CONSTANT-RAM  --  Sets all RAM cells to VALUE.


(defn constant-ram (mem value)
  (if (ramp mem)
      (ram value)
    (if (nlistp mem)
        mem
      (cons (constant-ram (car mem) value)
            (constant-ram (cdr mem) value)))))


(disable constant-ram)

;;;   LEMMAS
```

```
(prove-lemma memory-properp-if (rewrite)
  (implies
   (and (memory-properp n width a)
        (memory-properp n width b))
   (memory-properp n width (if c a b))))


(prove-lemma memory-okp-if (rewrite)
  (implies
   (and (memory-okp n width a)
        (memory-okp n width b))
   (memory-okp n width (if c a b))))


(prove-lemma memory-properp-constant-ram (rewrite)
  (implies
   (and (memory-properp n width mem)
        (properp value)
        (equal width (length value)))
   (memory-properp n width (constant-ram mem value)))
  ;;Hint
  ((enable memory-properp constant-ram)))


(prove-lemma memory-properp-after-write-mem1 (rewrite)
  (implies
   (and (memory-properp n width mem)
        (properp value)
        (equal width (length value))
        (equal n (length v-addr)))
   (memory-properp n width (write-mem1 v-addr mem value)))
  ;;Hint
  ((enable memory-properp length write-mem1)))


(prove-lemma memory-properp-after-write-mem (rewrite)
  (implies
   (and (memory-properp n width mem)
        (properp value)
        (equal width (length value))
        (equal n (length v-addr)))
   (memory-properp n width (write-mem v-addr mem value)))
  ;;Hint
  ((enable write-mem)))
```

```
(prove-lemma memory-okp-after-write-mem1 (rewrite)
  (implies
   (and (memory-okp n width mem)
        (bvp value)
        (equal width (length value))
        (equal n (length v-addr)))
   (memory-okp n width (write-mem1 v-addr mem value)))
  ;;Hint
  ((enable memory-okp length write-mem1)))


(prove-lemma memory-okp-after-write-mem (rewrite)
  (implies
   (and (memory-okp n width mem)
        (bvp value)
        (equal width (length value))
        (equal n (length v-addr)))
   (memory-okp n width (write-mem v-addr mem value)))
  ;;Hint
  ((enable write-mem)))


(prove-lemma v-iff-v-addr1-v-addr2-read-mem1-write-mem1 (rewrite)
  (implies
   (and (v-iff v-addr1 v-addr2)
        (ramp-mem1 v-addr2 mem)
        (equal (length v-addr1) (length v-addr2)))
   (equal (read-mem1 v-addr1 (write-mem1 v-addr2 mem value))
          value))
  ;;Hint
  ((enable v-iff memory-okp read-mem1 write-mem1 ramp-mem1)))


(prove-lemma v-iff-v-addr1-v-addr2-read-mem1-write-mem1-not-ram (rewrite)
  (implies
   (and (not (ramp-mem1 v-addr2 mem))
        (equal (length v-addr1) (length v-addr2)))
   (equal (read-mem1 v-addr1 (write-mem1 v-addr2 mem value))
          (read-mem1 v-addr1 mem)))
  ;;Hint
  ((enable v-iff memory-okp read-mem1 write-mem1 ramp-mem1)))


(prove-lemma not-v-iff-v-addr1-v-addr2-read-mem1-write-mem1 (rewrite)
  (implies
   (not (v-iff v-addr1 v-addr2))
   (equal (read-mem1 v-addr1 (write-mem1 v-addr2 mem value))
          (read-mem1 v-addr1 mem)))
  ;;Hint
  ((enable v-iff memory-okp read-mem1 write-mem1)))
```

```
(prove-lemma read-mem-write-mem ()
  (implies
   (equal (length v-addr1) (length v-addr2))
   (equal (read-mem v-addr1 (write-mem v-addr2 mem value))
          (if (and (v-iff v-addr1 v-addr2)
                   (ramp-mem v-addr2 mem))
              value
            (read-mem v-addr1 mem))))
  ;;Hint
  ((enable read-mem write-mem ramp-mem)))


(prove-lemma properp-read-mem1 (rewrite)
  (implies
   (memory-properp (length v-addr) size mem)
   (and (properp (read-mem1 v-addr mem))
        (equal (length (read-mem1 v-addr mem)) size)))
  ;;Hint
  ((enable memory-properp length read-mem1)))

;;;  This lemma is inapplicable because of the free variable SIZE.


(prove-lemma properp-read-mem ()
  (implies
   (memory-properp (length (reverse v-addr)) size mem)
   (and (properp (read-mem v-addr mem))
        (equal (length (read-mem v-addr mem)) size)))
  ;;Hint
  ((enable read-mem)
   (disable length-reverse)))


(prove-lemma properp-read-mem-32 (rewrite)
  (implies (memory-properp (length v-addr) 32 mem)
           (and (properp (read-mem v-addr mem))
                (equal (length (read-mem v-addr mem)) 32)))
  ;;Hint
  ((use (properp-read-mem (v-addr v-addr) (size 32) (mem mem)))))


(prove-lemma bvp-read-mem1 (rewrite)
  (implies
   (memory-okp (length v-addr) size mem)
   (and (bvp (read-mem1 v-addr mem))
        (equal (length (read-mem1 v-addr mem)) size)))
  ;;Hint
  ((enable memory-okp length read-mem1)))

;;;  This lemma is inapplicable because of the free variable SIZE.
```

```
(prove-lemma bvp-read-mem ()
  (implies
   (memory-okp (length (reverse v-addr)) size mem)
   (and (bvp (read-mem v-addr mem))
        (equal (length (read-mem v-addr mem)) size)))
  ;;Hint
  ((enable read-mem)
   (disable length-reverse)))


(prove-lemma bvp-read-mem-32 (rewrite)
  (implies (memory-okp (length v-addr) 32 mem)
           (and (bvp (read-mem v-addr mem))
                (equal (length (read-mem v-addr mem)) 32)))
  ;;Hint
  ((use (bvp-read-mem (v-addr v-addr) (size 32) (mem mem)))))


(prove-lemma all-ramp-mem->ramp-mem1 (rewrite)
  (implies
   (all-ramp-mem (length v-addr) mem)
   (ramp-mem1 v-addr mem))
  ;;Hint
  ((enable all-ramp-mem ramp-mem1 length)))


(prove-lemma all-ramp-mem->ramp-mem (rewrite)
  (implies
   (all-ramp-mem (length v-addr) mem)
   (ramp-mem v-addr mem))
  ;;Hint
  ((enable all-ramp-mem ramp-mem)))


(prove-lemma all-ramp-mem-after-write-mem1 ()
  (implies
   (and (all-ramp-mem n mem)
        (equal n (length v-addr)))
   (all-ramp-mem n (write-mem1 v-addr mem value)))
  ;;Hint
  ((enable all-ramp-mem write-mem1 length)))


(prove-lemma all-ramp-mem-after-write-mem (rewrite)
  (implies
   (and (all-ramp-mem n mem)
        (equal n (length v-addr)))
   (all-ramp-mem n (write-mem v-addr mem value)))
  ;;Hint
  ((use (all-ramp-mem-after-write-mem1 (v-addr (reverse v-addr))))
   (enable write-mem)))
```

```
(prove-lemma all-ramp-mem-constant-ram (rewrite)
  (equal (all-ramp-mem n (constant-ram mem value))
         (all-ramp-mem n mem))
  ;;Hint
  ((enable all-ramp-mem constant-ram)))
```

```
(prove-lemma memory-okp==>memory-properp (rewrite)
  (implies
   (memory-okp n m mem)
   (memory-properp n m mem))
  ;;Hint
  ((enable memory-okp memory-properp)))
```

14.26 "dual-port-ram.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    DUAL-PORT-RAM.EVENTS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;  This file contains a model of a dual-port ram:
;;;
;;;  (DUAL-PORT-RAM-VALUE bits address-lines args state)
;;;
;;;  returns the RAM output, i.e., the contents of the memory addressed by the
;;;  read-adress port.
;;;
;;;  (DUAL-PORT-RAM-STATE bits address-lines args state)
;;;
;;;  updates the state of the RAM.
;;;
;;;  The ARGS are assumed to be structured as follows:
;;;
;;;  0..(ADDRESS-LINES - 1)               -- A (read port) address.
;;;  ADDRESS-LINES..(2*ADDRESS-LINES - 1) -- B (write port) address.
;;;  (2*ADDRESS-LINES)                    -- WEN, active low.
;;;  remainder                            -- DATA lines.
;;;
;;;    WARNING -- This is a sequential model of what is essentially a
;;; level-sensitive device.  Note that this state-holding device has no clock
;;; input.  Spikes on WEN, or changes on B-ADDRESS while WEN is active may
;;; cause unanticipated changes in the memory state of the real device.
;;;
;;;    The dual-port RAM used in the register file of the FM9001 is surrounded
;;;  by sequential logic that ensures that setup and hold constraints are met.
;;;  See the file "regfile.events".
```

```
(defn dual-port-ram-value (bits address-lines args state)
  (let ((a-address (subrange args 0 (sub1 address-lines)))
        (b-address (subrange args address-lines
                             (sub1 (times 2 address-lines))))
        (wen (nth (times 2 address-lines) args)))
    ;;  If the read address is unknown, or the device is potentially write
    ;;  enabled and there is a potential write at the read address, then read
    ;;  out X's.  Otherwise, read out the vector from the STATE.
    (if (or (not (bvp a-address))
            (and (not (equal wen t))
                 (or (not (bvp b-address))
                     (equal a-address b-address))))
        (make-list bits (x))
      (let ((val (read-mem a-address state)))
        (if (and (properp val)
                 (equal (length val) bits))
            val
          ;; Return an unknown proper list of the right length if we don't read
          ;; a proper list of the right length.
          (make-list bits (x)))))))


(defn dual-port-ram-state (bits address-lines args state)
  (let ((b-address (subrange args address-lines
                             (sub1 (times 2 address-lines))))
        (wen (nth (times 2 address-lines) args))
        ;; Use SUBRANGE instead of RESTN so that we are guaranteed
        ;; that this argument has the right length and is a PROPERP.
        ;; Note that we use bits below rather than (length args)
        ;; in order to ensure that data has the right length.
        (data
         (subrange args
                   (add1 (times 2 address-lines))
                   (plus (times 2 address-lines) bits))))
    ;;  If WEN is solidly high, do nothing.
    (if (equal wen t)
        state
      ;;  There is a potential write.  If the address is unknown, wipe out the
      ;;  state.
      (if (not (bvp b-address))
          (constant-ram state (make-list bits (x)))
        ;;  If WEN is solidly low, update the state with data, otherwise X out
        ;;  the addressed entry.
        (if (equal wen f)
            (write-mem b-address state data)
          (write-mem b-address state (make-list bits (x))))))))

;;;  LEMMAS
```

```
(prove-lemma properp-length-dual-port-ram-value (rewrite)
  (and
   (properp (dual-port-ram-value bits address-lines args state))
   (equal (length (dual-port-ram-value bits address-lines args state))
          (fix bits))))
```

14.27 "fm9001-memory.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    FM9001-MEMORY.EVENTS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;   This is a model of a generic memory for the FM9001.  Besides the
;;;   state component, the memory takes as input a STROBE (active low), a
;;;   read/write line (low to write), and address and data lines.
;;;
;;;   The state component consists of the memory contents, a memory control
;;;   state (see below), a "clock" (see below), a "count" used for DTACK
;;;   scheduling, a flag indicating that DTACK has been asserted, the last
;;;   value of RW- and the last address and data inputs.
;;;
;;;   Address lines, and data lines on a write, must be stable for one cycle
;;;   before the strobe is activated, and one cycle after the strobe is
;;;   released.  On a write, the same setup/hold constraints also must be met
;;;   by the low pulse on RW-.
;;;
;;;   Memory control states:
;;;
;;;   0 -- The quiet state.
;;;   1 -- Read wait.
;;;   2 -- Write wait.
;;;   x -- Error.
;;;
;;;   The "clock" is an oracle that specifies the wait for DTACK.  Whenever the
;;;   STROBE is activated, then (CAR clock) is the number of memory delays for
;;;   this memory operation, and the new "clock" becomes (CDR clock).  For
;;;   simulation purposes we normally set this to 0 to provide minimum delays
;;;   since both the CAR and CDR of 0 are 0.
```

```
(defn next-memory-state (state strobe- rw- address data)
  (let ((mem            (car state))
        (cntl           (cadr state))
        (clock          (caddr state))
        (count          (cadddr state))
        (dtack-asserted (caddddr state))
        (last-rw-       (cadddddr state))
        (last-address   (caddddddr state))
        (last-data      (cadddddddr state)))

    (let ((rw-address-data
           (list (threefix rw-)
                 (v-threefix address)
                 (if (boolp rw-)
                     (if rw-
                         (v-threefix last-data)
                       (v-threefix data))
                   (make-list (length data) (x))))))
      (let
        ((reset       (list* mem 0 clock 0 t rw-address-data))
         (start-read  (list* mem 1 (cdr clock)
                             (sub1 (car clock)) (zerop (car clock))
                             rw-address-data))
         (start-write (list* mem 2 (cdr clock)
                             (sub1 (car clock)) (zerop (car clock))
                             rw-address-data))
         (read-error  (list* mem 3 (cdr clock)
                             (sub1 (car clock)) (zerop (car clock))
                             rw-address-data))
         (write-error (list* (constant-ram mem (make-list 32 (x)))
                             3 (cdr clock)
                             (sub1 (car clock)) (zerop (car clock))
                             rw-address-data))
         (reset-error  (list* (constant-ram mem (make-list 32 (x)))
                              0 clock 0 t
                              rw-address-data))
         (finish-write (list* (write-mem address mem data) 0 clock 0 t
                              rw-address-data))
         (read-wait  (list* mem 1 clock (sub1 count) (zerop count)
                            rw-address-data))
         (write-wait (list* mem 2 clock (sub1 count) (zerop count)
                            rw-address-data))
         (error-wait (list* mem 3 clock (sub1 count) (zerop count)
                            rw-address-data)))

        (let ((bvp-equal-address (and (bvp address)
                                      (bvp last-address)
                                      (equal address last-address)))
              (bvp-equal-data (and (bvp data)
                                   (bvp last-data)
                                   (equal data last-data))))

          (if (and (boolp strobe-) (boolp rw-))
```

```
              (case cntl
                (0 (if strobe-
                       reset
                     (if rw-
                         (if (and last-rw- (boolp last-rw-))
                             (if bvp-equal-address
                                 start-read
                               read-error)
                           write-error)
                       (if (and (not last-rw-) ;Subtle
                                bvp-equal-address
                                bvp-equal-data)
                           start-write
                         write-error))))

                (1 (if strobe-
                       (if rw-
                           reset
                         reset-error)
                     (if rw-
                         (if bvp-equal-address
                             read-wait
                           read-error)
                       write-error)))

                (2 (if strobe-
                       (if rw-
                           reset-error
                         (if (and bvp-equal-address
                                  bvp-equal-data
                                  (zerop count))
                             finish-write
                           reset-error))
                     (if rw-
                         write-error
                       (if (and bvp-equal-address
                                bvp-equal-data)
                           write-wait
                         write-error))))

                (otherwise (if strobe-
                               (if rw-
                                   reset
                                 reset-error)
                             (if rw-
                                 error-wait
                               write-error))))

            reset-error))))))

(disable next-memory-state)
```

```
(defn memory-value (state strobe- rw- address data)
  (let ((mem            (car state))
        (cntl           (cadr state))
        (clock          (caddr state))
        (count          (cadddr state))
        (dtack-asserted (caddddr state))
        (last-rw-       (cadddddr state))
        (last-address   (caddddddr state))
        (last-data      (cadddddddr state)))
    (let ((x-vector    (make-list (length data) (x)))
          (z-vector    (make-list (length data) (z))))

      (let ((unknown        (cons (x) x-vector))
            (default        (cons (x) z-vector))
            (read-wait      (cons t x-vector))
            (write-wait     (cons t z-vector))
            (dtack-0        (cons (if* (zerop (car clock)) f t)
                                  (if* rw- x-vector z-vector)))
            (dtack-read     (cons f
                                  (if* dtack-asserted
                                      (read-mem address mem)
                                    x-vector)))
            (dtack-read-default  (cons f x-vector))
            (dtack-write-default  (cons f z-vector)))

        (let ((bvp-equal-address (and* (equal address last-address)
                                       (and* (bvp address)
                                             (bvp last-address)))))


        (if* (and* (boolp strobe-) (boolp rw-))

            (case cntl
              (0 (if* strobe-
                     default
                   dtack-0))

              (1 (if* strobe-
                     default
                   (if* rw-
                       (if* bvp-equal-address
                           (if* (zerop count)
                               dtack-read
                             read-wait)
                         (if* (zerop count)
                             dtack-read-default
                           read-wait))
                     (if* (zerop count)
                         dtack-write-default
                       write-wait))))

              (2 (if* strobe-
                     default
                   (if* (zerop count)
```

```
                     dtack-write-default
                   write-wait)))

            (otherwise (if* strobe-
                            default
                          (if* rw-
                              (if* (zerop count)
                                  dtack-read-default
                                read-wait)
                            (if* (zerop count)
                                dtack-write-default
                              write-wait)))))

        unknown))))))


(disable memory-value)

;;;  A couple of helper definitions to make it easy to call
;;;  the FM9001 memory.


(defn mem-value (args state)
  (let ((rw-     (car args))
        (strobe- (cadr args))
        (address (subrange args 2 33))
        (data    (subrange args 34 65)))
    (memory-value state strobe- rw- address data)))


(disable mem-value)


(defn mem-state (args state)
  (let ((rw-     (car args))
        (strobe- (cadr args))
        (address (subrange args 2 33))
        (data    (subrange args 34 65)))
    (next-memory-state state strobe- rw- address data)))


(disable mem-state)

;;;    Lemmas
```

```
(prove-lemma properp-length-memory-value (rewrite)
  (implies
   (and (memory-properp 32 32 (car state))
        (equal (length address) 32)
        (equal (length data) 32))
   (and (properp (cdr (memory-value state strobe rw- address data)))
        (equal (length (cdr (memory-value state strobe rw- address data)))
               32)))
  ;;Hint
  ((enable memory-value)
   (disable *1*make-list)))


(prove-lemma equal-memory-value (rewrite)
  (implies
   (equal (length data1) (length data2))
   (equal (equal (memory-value state strobe rw- address data1)
                 (memory-value state strobe rw- address data2))
          t))
  ;;Hint
  ((enable memory-value)))

;;;   This "induction" makes the proof of the next lemma tractable.  This is
;;;   really a sneaky way to make the prover case-split in the desired way.
```

```
(defn next-memory-state$induction (state strobe- rw- address data)
  (let ((mem            (car state))
        (cntl           (cadr state))
        (clock          (caddr state))
        (count          (cadddr state))
        (dtack-asserted (caddddr state))
        (last-rw-       (cadddddr state))
        (last-address   (caddddddr state))
        (last-data      (cadddddddr state)))

    (let ((rw-address-data
           (list (threefix rw-)
                 (v-threefix address)
                 (if (boolp rw-)
                     (if rw-
                         (v-threefix last-data)
                       (v-threefix data))
                   (make-list (length data) (x))))))
      (let
        ((reset       (list* mem 0 clock 0 t rw-address-data))
         (start-read  (list* mem 1 (cdr clock)
                             (sub1 (car clock)) (zerop (car clock))
                             rw-address-data))
         (start-write (list* mem 2 (cdr clock)
                             (sub1 (car clock)) (zerop (car clock))
                             rw-address-data))
         (read-error  (list* mem 3 (cdr clock)
                             (sub1 (car clock)) (zerop (car clock))
                             rw-address-data))
         (write-error (list* (constant-ram mem (make-list 32 (x)))
                             3 (cdr clock)
                             (sub1 (car clock)) (zerop (car clock))
                             rw-address-data))
         (reset-error  (list* (constant-ram mem (make-list 32 (x)))
                              0 clock 0 t
                              rw-address-data))
         (finish-write (list* (write-mem address mem data) 0 clock 0 t
                              rw-address-data))
         (read-wait  (list* mem 1 clock (sub1 count) (zerop count)
                            rw-address-data))
         (write-wait (list* mem 2 clock (sub1 count) (zerop count)
                            rw-address-data))
         (error-wait (list* mem 3 clock (sub1 count) (zerop count)
                            rw-address-data)))

        (let ((bvp-equal-address (and (bvp address)
                                      (bvp last-address)
                                      (equal address last-address)))
              (bvp-equal-data (and (bvp data)
                                   (bvp last-data)
                                   (equal data last-data))))

          (if (and (boolp strobe-) (boolp rw-))
```

```
              (case cntl
                (0 (if strobe-
                       reset
                     (if rw-
                         (if (and last-rw- (boolp last-rw-))
                             (if bvp-equal-address
                                 start-read
                               read-error)
                           write-error)
                       (if (and (not last-rw-) ;Subtle
                                bvp-equal-address
                                bvp-equal-data)
                           start-write
                         write-error))))

                (1 (if strobe-
                       (if rw-
                           reset
                         reset-error)
                     (if rw-
                         (if bvp-equal-address
                             read-wait
                           read-error)
                       write-error)))

                (2 (if strobe-
                       (if rw-
                           reset-error
                         (if (and bvp-equal-address
                                  bvp-equal-data
                                  (zerop count))
                             finish-write
                           reset-error))
                     (if rw-
                         write-error
                       (if (and bvp-equal-address
                                bvp-equal-data)
                           write-wait
                         write-error))))

                (otherwise (if strobe-
                               (if rw-
                                   reset
                                 reset-error)
                             (if rw-
                                 error-wait
                               write-error))))

            (if (not (and (boolp strobe-) (boolp rw-)))
                reset-error
              (next-memory-state$induction (sub1 state)
                                           strobe- rw- address data)))))))
  ;;Hint
  ((lessp (count state))))
```

```
(prove-lemma properp-length-next-memory-state (rewrite)
  (and (properp (next-memory-state state strobe- rw- address data))
       (equal (length (next-memory-state state strobe- rw- address data)) 8))
  ;;Hint
  ((induct (next-memory-state$induction
            state strobe- rw- address data))
   (enable next-memory-state)))
```

14.28 "tree-number.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    TREE-NUMBER.EVENTS
;;;
;;;    (TREE-NUMBER tree) returns a unique (we think) number for each
;;;    equivalence class of trees with the same CONS structure.  We use this
;;;    to give unique, numerical indices to modules created by tree-based
;;;    module generators.  We never proved that TREE-NUMBER yields a unique
;;;    encoding, but our netlist predicates would fail if it were ever the case
;;;    that non-unique encodings were produced.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    This definition of TREE-NUMBER has the property that a balanced tree
;;;    with N leaves has (TREE-NUMBER TREE) = N.   In the binary encoding of
;;;    the tree, each "full" level (whether full of CONS cells or leaves) is
;;;    encoded as T.  Partially empty levels are encoded by F, followed by a
;;;    bit string that encodes with T and F those locations that are filled and
;;;    empty respectively.


(defn fix-breadth-tree-stack (stack n)
  (if (nlistp stack)
      nil
    (cons
     (append (make-list n f) (car stack))
     (fix-breadth-tree-stack (cdr stack) (times 2 n)))))


(defn breadth-tree (tree stack n)
  (if (nlistp tree)
      (cons (cons t (if (nlistp stack)
                        (make-list n f)
                      (car stack)))
            (fix-breadth-tree-stack (cdr stack) 2))
    (cons
     (cons t (if (nlistp stack)
                 (make-list n f)
               (car stack)))
     (breadth-tree (cdr tree)
                   (breadth-tree (car tree) (cdr stack) (times 2 n))
                   (add1 (times 2 n))))))
```

```
(defn collect-breadth-tree (stack n)
  (if (nlistp stack)
      nil
    (if (equal (car stack) (make-list n t))
        (cons t (collect-breadth-tree (cdr stack) (times 2 n)))
      (cons f (append (car stack)
                      (collect-breadth-tree (cdr stack) (times 2 n)))))))


(defn tree-number (tree)
  (quotient (add1
             (v-to-nat (collect-breadth-tree (breadth-tree tree nil 0) 1)))
            2))


(disable tree-number)

#|
Problem:  Prove that TREE-NUMBER yields a unique encoding for isomorphic trees.
That is, given


(defn isomorphic (x y)
  (if (or (nlistp x) (nlistp y))
      (and (nlistp x) (nlistp y))
    (and (isomorphic (car x) (car y))
         (isomorphic (cdr x) (cdr y)))))

then show


(iff (isomorphic t1 t2)
     (equal (tree-number t1)
            (tree-number t2))).

|#
```

14.29 "f-functions.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    F-FUNCTIONS.EVENTS
;;;
;;;  Definitions of basic 3- and 4-valued specifications for DUAL-EVAL.
;;;  This file also includes some 3- and 4-valued vector specifications.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;   We begin by introducing primitive functions for 4-state logic.


(defn f-buf (a) (threefix a))


(defn f-and (a b)
  (if (or (equal a f) (equal b f))
      f
    (if (and (equal a t) (equal b t))
        t
      (x))))


(defn f-and3 (a b c) (f-and a (f-and b c)))

(defn f-and4 (a b c d) (f-and a (f-and b (f-and c d))))


(defn f-not (a)
  (if (boolp a)
      (not a)
    (x)))


(defn f-nand (a b) (f-not (f-and a b)))

(defn f-nand3 (a b c) (f-not (f-and a (f-and b c))))

(defn f-nand4 (a b c d) (f-not (f-and a (f-and b (f-and c d)))))

(defn f-nand5 (a b c d e) (f-not (f-and a (f-and b (f-and c (f-and d e))))))

(defn f-nand6 (a b c d e g)
  (f-not (f-and a (f-and b (f-and c (f-and d (f-and e g)))))))

(defn f-nand8 (a b c d e g h i)
  (f-not
   (f-and a (f-and b (f-and c (f-and d (f-and e (f-and g (f-and h i)))))))))
```

```
(defn f-or (a b)
  (if (or (equal a t) (equal b t))
      t
    (if (and (equal a f) (equal b f))
        f
      (x))))


(defn f-or3 (a b c) (f-or a (f-or b c)))

(defn f-or4 (a b c d) (f-or a (f-or b (f-or c d))))


(defn f-nor (a b) (f-not (f-or a b)))

(defn f-nor3 (a b c) (f-not (f-or a (f-or b c))))

(defn f-nor4 (a b c d) (f-not (f-or a (f-or b (f-or c d)))))

(defn f-nor5 (a b c d e) (f-not (f-or a (f-or b (f-or c (f-or d e))))))

(defn f-nor6 (a b c d e g)
  (f-not (f-or a (f-or b (f-or c (f-or d (f-or e g)))))))

(defn f-nor8 (a b c d e g h i)
  (f-not
   (f-or a (f-or b (f-or c (f-or d (f-or e (f-or g (f-or h i)))))))))


(defn f-xor (a b)
  (if (equal a t)
      (f-not b)
    (if (equal a f)
        (threefix b)
      (x))))


(defn f-xor3 (a b c) (f-xor a (f-xor b c)))


(defn f-equv (a b)
  (if (equal a t)
      (threefix b)
    (if (equal a f)
        (f-not b)
      (x))))


(defn f-equv3 (a b c) (f-equv a (f-xor b c)))
```

```
(defn f-if (c a b)
  (if (equal c t)
      (threefix a)
    (if (equal c f)
        (threefix b)
      (x))))


(defn ft-buf (c a)
  (if (equal c t)
      (threefix a)
    (if (equal c f)
        (z)
      (x))))


(defn ft-wire (a b)
  (if (equal a b)
     (fourfix a)
    (if (equal a (z))
        (fourfix b)
      (if (equal b (z))
          (fourfix a)
        (x)))))


(defn f-pullup (a)
  (if (equal a (z))
      t
    (threefix a)))

;;;  We need this because we later use DEFN-TO-MODULE to generate modules
;;;  containing references to AO2.


(defn f$ao2 (a b c d) (f-nor (f-and a b) (f-and c d)))
```

```
(prove-lemma expand-f-functions (rewrite)
  (and
   (equal (f-buf a) (threefix a))
   (equal (f-and a b)
          (if (or (equal a f) (equal b f))
              f
            (if (and (equal a t) (equal b t))
                t
              (x))))
   (equal (f-and3 a b c) (f-and a (f-and b c)))
   (equal (f-and4 a b c d) (f-and a (f-and b (f-and c d))))
   (equal (f-not a)
          (if (boolp a)
              (not a)
            (x)))
   (equal (f-nand a b) (f-not (f-and a b)))
   (equal (f-nand3 a b c) (f-not (f-and a (f-and b c))))
   (equal (f-nand4 a b c d) (f-not (f-and a (f-and b (f-and c d)))))
   (equal (f-or a b)
          (if (or (equal a t) (equal b t))
              t
            (if (and (equal a f) (equal b f))
                f
              (x))))
   (equal (f-nand5 a b c d e)
          (f-not (f-and a (f-and b (f-and c (f-and d e))))))
   (equal (f-nand6 a b c d e g)
          (f-not (f-and a (f-and b (f-and c (f-and d (f-and e g)))))))
   (equal (f-nand8 a b c d e g h i)
          (f-not
           (f-and
            a (f-and b (f-and c (f-and d (f-and e (f-and g (f-and h i)))))))))
   (equal (f-or3 a b c) (f-or a (f-or b c)))
   (equal (f-or4 a b c d) (f-or a (f-or b (f-or c d))))
   (equal (f-nor a b) (f-not (f-or a b)))
   (equal (f-nor3 a b c) (f-not (f-or a (f-or b c))))
   (equal (f-nor4 a b c d) (f-not (f-or a (f-or b (f-or c d)))))
   (equal (f-nor5 a b c d e)
          (f-not (f-or a (f-or b (f-or c (f-or d e))))))
   (equal (f-nor6 a b c d e g)
          (f-not (f-or a (f-or b (f-or c (f-or d (f-or e g)))))))
   (equal (f-nor8 a b c d e g h i)
          (f-not
           (f-or
            a (f-or b (f-or c (f-or d (f-or e (f-or g (f-or h i)))))))))
   (equal (f-xor a b)
          (if (equal a t)
              (f-not b)
            (if (equal a f)
                (threefix b)
              (x))))
   (equal (f-xor3 a b c) (f-xor a (f-xor b c)))
   (equal (f-equv a b)
          (if (equal a t)
```

```
              (threefix b)
            (if (equal a f)
                (f-not b)
              (x))))
   (equal (f-equv3 a b c) (f-equv a (f-xor b c)))
   (equal (f-if c a b)
          (if (equal c t)
              (threefix a)
            (if (equal c f)
                (threefix b)
              (x))))
   (equal (ft-buf c a)
          (if (equal c t)
              (threefix a)
            (if (equal c f)
                (z)
              (x))))
   (equal (ft-wire a b)
          (if (equal a b)
              (fourfix a)
            (if (equal a (z))
                (fourfix b)
              (if (equal b (z))
                  (fourfix a)
                (x)))))
   (equal (f-pullup a)
          (if (equal a (z))
              t
            (threefix a)))))


(disable expand-f-functions)

;;;  This lemma allows us to use F-BUF as a no-op.


(prove-lemma f-buf-lemma (rewrite)
  (implies
   (boolp x)
   (equal (f-buf x)
          x)))

;;;   Some facts for those times when various "F-functions" are disabled.
```

```
(prove-lemma f-and-rewrite (rewrite)
  (and
   (not (f-and f x))
   (not (f-and x f))
   (equal (f-and x t)
          (f-buf x))
   (equal (f-and t x)
          (f-buf x))
   (implies
    (and x y (boolp x) (boolp y))
    (equal (f-and x y)
           t))))


(prove-lemma f-or-rewrite (rewrite)
  (and
   (equal (f-or f f)
          f)
   (implies
    (and x (boolp x))
    (equal (f-or x y) t))
   (implies
    (and x (boolp x))
    (equal (f-or y x) t))))


(prove-lemma f-not-rewrite (rewrite)
  (and
   (equal (f-not t) f)
   (equal (f-not f) t)))


(prove-lemma ft-buf-rewrite (rewrite)
  (and
   (equal (ft-buf t x)
          (threefix x))
   (equal (ft-buf f x)
          (z))
   (equal (ft-buf c (threefix x))
          (ft-buf c x))
   (equal (ft-buf (x) x)
          (x))))


(prove-lemma f-if-rewrite (rewrite)
  (and
   (equal (f-if t a b)
          (f-buf a))
   (equal (f-if f a b)
          (f-buf b))))
```

```
(prove-lemma ft-wire-rewrite (rewrite)
  (and
   (equal (ft-wire (x) x)
          (x))
   (equal (ft-wire x (x))
          (x))
   (equal (ft-wire x (z))
          (fourfix x))
   (equal (ft-wire (z) x)
          (fourfix x))))


(prove-lemma f-pullup-rewrite (rewrite)
  (and
   (equal (f-pullup t) t)
   (equal (f-pullup f) f)
   (equal (f-pullup (x)) (x))
   (equal (f-pullup (z)) t)))

;;; Leading up to a rewrite rule to get rid of extra F-BUF's.


(prove-lemma threefix-help-lemma (rewrite)
  (and
   (equal (threefix f) f)
   (equal (threefix t) t)
   (equal (threefix (x)) (x))
   (equal (threefix (f-buf x))    (threefix x))
   (equal (threefix (f-not x))    (f-not x))
   (equal (threefix (f-and x y))  (f-and x y))
   (equal (threefix (f-or x y))   (f-or x y))
   (equal (threefix (f-xor x y))  (f-xor x y))
   (equal (threefix (f-equv x y)) (f-equv x y))))


(disable threefix-help-lemma)


(prove-lemma f-not-f-not=f-buf (rewrite)
  (equal (f-not (f-not x))
         (f-buf x)))


(disable f-not-f-not=f-buf)
```

```
(prove-lemma f-buf-delete-lemmas (rewrite)
  (and
   (equal (f-buf (f-buf a))
          (f-buf a))
   (equal (f-buf (f-not a))
          (f-not a))
   (equal (f-buf (f-nand a b))
          (f-nand a b))
   (equal (f-buf (f-nand3 a b c))
          (f-nand3 a b c))
   (equal (f-buf (f-nand4 a b c d))
          (f-nand4 a b c d))
   (equal (f-buf (f-nand5 a b c d e))
          (f-nand5 a b c d e))
   (equal (f-buf (f-nand6 a b c d e g))
          (f-nand6 a b c d e g))
   (equal (f-buf (f-nand8 a b c d e g h i))
          (f-nand8 a b c d e g h i))
   (equal (f-buf (f-or a b))
          (f-or a b))
   (equal (f-buf (f-or3 a b c))
          (f-or3 a b c))
   (equal (f-buf (f-or4 a b c d))
          (f-or4 a b c d))
   (equal (f-buf (f-xor a b))
          (f-xor a b))
   (equal (f-buf (f-xor3 a b c))
          (f-xor3 a b c))
   (equal (f-buf (f-equv a b))
          (f-equv a b))
   (equal (f-buf (f-equv3 a b c))
          (f-equv3 a b c))
   (equal (f-buf (f-and a b))
          (f-and a b))
   (equal (f-buf (f-and3 a b c))
          (f-and3 a b c))
   (equal (f-buf (f-and4 a b c d))
          (f-and4 a b c d))
   (equal (f-buf (f-nor a b))
          (f-nor a b))
   (equal (f-buf (f-nor3 a b c))
          (f-nor3 a b c))
   (equal (f-buf (f-nor4 a b c d))
          (f-nor4 a b c d))
   (equal (f-buf (f-nor5 a b c d e))
          (f-nor5 a b c d e))
   (equal (f-buf (f-nor6 a b c d e g))
          (f-nor6 a b c d e g))
   (equal (f-buf (f-nor8 a b c d e g h i))
          (f-nor8 a b c d e g h i))
   (equal (f-buf (f-if c a b))
          (f-if c a b)))
  ((enable threefix-help-lemma)))
```

```
(disable f-buf-delete-lemmas)


(prove-lemma f-gate-threefix-congruence-lemmas$help (rewrite)
  (and
   (equal (f-buf (threefix a))
          (f-buf a))
   (equal (f-not (threefix a))
          (f-not a))
   (equal (f-or (threefix a) b)
          (f-or a b))
   (equal (f-or a (threefix b))
          (f-or a b))
   (equal (f-xor (threefix a) b)
          (f-xor a b))
   (equal (f-xor a (threefix b))
          (f-xor a b))
   (equal (f-equv (threefix a) b)
          (f-equv a b))
   (equal (f-equv a (threefix b))
          (f-equv a b))
   (equal (f-and (threefix a) b)
          (f-and a b))
   (equal (f-and a (threefix b))
          (f-and a b))
   (equal (f-if (threefix c) a b)
          (f-if c a b))
   (equal (f-if c (threefix a) b)
          (f-if c a b))
   (equal (f-if c a (threefix b))
          (f-if c a b))))


(disable f-gate-threefix-congruence-lemmas$help)
```

```
(prove-lemma f-gate-threefix-congruence-lemmas (rewrite)
  (and
   (equal (f-buf (threefix a))
          (f-buf a))

   (equal (f-not (threefix a))
          (f-not a))

   (equal (f-nand (threefix a) b)
          (f-nand a b))
   (equal (f-nand a (threefix b))
          (f-nand a b))

   (equal (f-nand3 (threefix a) b c)
          (f-nand3 a b c))
   (equal (f-nand3 a (threefix b) c)
          (f-nand3 a b c))
   (equal (f-nand3 a (threefix b) c)
          (f-nand3 a b c))

   (equal (f-nand4 (threefix a) b c d)
          (f-nand4 a b c d))
   (equal (f-nand4 a (threefix b) c d)
          (f-nand4 a b c d))
   (equal (f-nand4 a b (threefix c) d)
          (f-nand4 a b c d))
   (equal (f-nand4 a b c (threefix d))
          (f-nand4 a b c d))

   (equal (f-nand5 (threefix a) b c d e)
          (f-nand5 a b c d e))
   (equal (f-nand5 a (threefix b) c d e)
          (f-nand5 a b c d e))
   (equal (f-nand5 a b (threefix c) d e)
          (f-nand5 a b c d e))
   (equal (f-nand5 a b c (threefix d) e)
          (f-nand5 a b c d e))
   (equal (f-nand5 a b c d (threefix e))
          (f-nand5 a b c d e))

   (equal (f-nand6 (threefix a) b c d e g)
          (f-nand6 a b c d e g))
   (equal (f-nand6 a (threefix b) c d e g)
          (f-nand6 a b c d e g))
   (equal (f-nand6 a b (threefix c) d e g)
          (f-nand6 a b c d e g))
   (equal (f-nand6 a b c (threefix d) e g)
          (f-nand6 a b c d e g))
   (equal (f-nand6 a b c d (threefix e) g)
          (f-nand6 a b c d e g))
   (equal (f-nand6 a b c d e (threefix g))
          (f-nand6 a b c d e g))

   (equal (f-nand8 (threefix a) b c d e g h i)
```

```
       (f-nand8 a b c d e g h i))
(equal (f-nand8 a (threefix b) c d e g h i)
       (f-nand8 a b c d e g h i))
(equal (f-nand8 a b (threefix c) d e g h i)
       (f-nand8 a b c d e g h i))
(equal (f-nand8 a b c (threefix d) e g h i)
       (f-nand8 a b c d e g h i))
(equal (f-nand8 a b c d (threefix e) g h i)
       (f-nand8 a b c d e g h i))
(equal (f-nand8 a b c d e (threefix g) h i)
       (f-nand8 a b c d e g h i))
(equal (f-nand8 a b c d e g (threefix h) i)
       (f-nand8 a b c d e g h i))
(equal (f-nand8 a b c d e g h (threefix i))
       (f-nand8 a b c d e g h i))

(equal (f-or (threefix a) b)
       (f-or a b))
(equal (f-or a (threefix b))
       (f-or a b))

(equal (f-or3 (threefix a) b c)
       (f-or3 a b c))
(equal (f-or3 a (threefix b) c)
       (f-or3 a b c))
(equal (f-or3 a (threefix b) c)
       (f-or3 a b c))

(equal (f-or4 (threefix a) b c d)
       (f-or4 a b c d))
(equal (f-or4 a (threefix b) c d)
       (f-or4 a b c d))
(equal (f-or4 a b (threefix c) d)
       (f-or4 a b c d))
(equal (f-or4 a b c (threefix d))
       (f-or4 a b c d))

(equal (f-xor (threefix a) b)
       (f-xor a b))
(equal (f-xor a (threefix b))
       (f-xor a b))

(equal (f-xor3 (threefix a) b c)
       (f-xor3 a b c))
(equal (f-xor3 a (threefix b) c)
       (f-xor3 a b c))
(equal (f-xor3 a (threefix b) c)
       (f-xor3 a b c))

(equal (f-equv (threefix a) b)
       (f-equv a b))
(equal (f-equv a (threefix b))
       (f-equv a b))

(equal (f-equv3 (threefix a) b c)
```

```
         (f-equv3 a b c))
    (equal (f-equv3 a (threefix b) c)
         (f-equv3 a b c))
    (equal (f-equv3 a (threefix b) c)
         (f-equv3 a b c))

    (equal (f-and (threefix a) b)
         (f-and a b))
    (equal (f-and a (threefix b))
         (f-and a b))

    (equal (f-and3 (threefix a) b c)
         (f-and3 a b c))
    (equal (f-and3 a (threefix b) c)
         (f-and3 a b c))
    (equal (f-and3 a (threefix b) c)
         (f-and3 a b c))

    (equal (f-and4 (threefix a) b c d)
         (f-and4 a b c d))
    (equal (f-and4 a (threefix b) c d)
         (f-and4 a b c d))
    (equal (f-and4 a b (threefix c) d)
         (f-and4 a b c d))
    (equal (f-and4 a b c (threefix d))
         (f-and4 a b c d))

    (equal (f-nor (threefix a) b)
         (f-nor a b))
    (equal (f-nor a (threefix b))
         (f-nor a b))

    (equal (f-nor3 (threefix a) b c)
         (f-nor3 a b c))
    (equal (f-nor3 a (threefix b) c)
         (f-nor3 a b c))
    (equal (f-nor3 a (threefix b) c)
         (f-nor3 a b c))

    (equal (f-nor4 (threefix a) b c d)
         (f-nor4 a b c d))
    (equal (f-nor4 a (threefix b) c d)
         (f-nor4 a b c d))
    (equal (f-nor4 a b (threefix c) d)
         (f-nor4 a b c d))
    (equal (f-nor4 a b c (threefix d))
         (f-nor4 a b c d))

    (equal (f-nor5 (threefix a) b c d e)
         (f-nor5 a b c d e))
    (equal (f-nor5 a (threefix b) c d e)
         (f-nor5 a b c d e))
    (equal (f-nor5 a b (threefix c) d e)
         (f-nor5 a b c d e))
    (equal (f-nor5 a b c (threefix d) e)
```

```
          (f-nor5 a b c d e))
   (equal (f-nor5 a b c d (threefix e))
          (f-nor5 a b c d e))

   (equal (f-nor6 (threefix a) b c d e g)
          (f-nor6 a b c d e g))
   (equal (f-nor6 a (threefix b) c d e g)
          (f-nor6 a b c d e g))
   (equal (f-nor6 a b (threefix c) d e g)
          (f-nor6 a b c d e g))
   (equal (f-nor6 a b c (threefix d) e g)
          (f-nor6 a b c d e g))
   (equal (f-nor6 a b c d (threefix e) g)
          (f-nor6 a b c d e g))
   (equal (f-nor6 a b c d e (threefix g))
          (f-nor6 a b c d e g))

   (equal (f-nor8 (threefix a) b c d e g h i)
          (f-nor8 a b c d e g h i))
   (equal (f-nor8 a (threefix b) c d e g h i)
          (f-nor8 a b c d e g h i))
   (equal (f-nor8 a b (threefix c) d e g h i)
          (f-nor8 a b c d e g h i))
   (equal (f-nor8 a b c (threefix d) e g h i)
          (f-nor8 a b c d e g h i))
   (equal (f-nor8 a b c d (threefix e) g h i)
          (f-nor8 a b c d e g h i))
   (equal (f-nor8 a b c d e (threefix g) h i)
          (f-nor8 a b c d e g h i))
   (equal (f-nor8 a b c d e g (threefix h) i)
          (f-nor8 a b c d e g h i))
   (equal (f-nor8 a b c d e g h (threefix i))
          (f-nor8 a b c d e g h i))

   (equal (f-if (threefix c) a b)
          (f-if c a b))
   (equal (f-if c (threefix a) b)
          (f-if c a b))
   (equal (f-if c a (threefix b))
          (f-if c a b)))
  ;;Hint
  ((enable f-gate-threefix-congruence-lemmas$help)
   (disable f-buf f-not f-or f-and f-xor f-equv f-if threefix)))


(disable f-gate-threefix-congruence-lemmas)

;;;  A 4-valued gate theory.
```

```
(deftheory f-gates
  (
   f-buf f-not
   f-nand f-nand3 f-nand4 f-nand5 f-nand6 f-nand8
   f-or f-or3 f-or4
   f-xor f-xor3
   f-equv f-equv3
   f-and f-and3 f-and4
   f-nor f-nor3 f-nor4 f-nor5 f-nor6 f-nor8
   f-if ft-buf ft-wire f-pullup))

;;;   When the F-GATE theory is disabled, the following lemma lets us
;;;   substitute a B-GATE for each F-GATE under assumptions that the inputs are
;;;   Boolean.
```

```
(prove-lemma f-gates=b-gates (rewrite)
  (and
   (implies (boolp a) (equal (f-buf a) (b-buf a)))
   (implies (boolp a) (equal (f-not a) (b-not a)))
   (implies (and (boolp a) (boolp b)) (equal (f-nand a b) (b-nand a b)))
   (implies (and (boolp a) (boolp b) (boolp c))
            (equal (f-nand3 a b c) (b-nand3 a b c)))
   (implies (and (boolp a) (boolp b) (boolp c) (boolp d))
            (equal (f-nand4 a b c d) (b-nand4 a b c d)))
   (implies (and (boolp a) (boolp b) (boolp c) (boolp d) (boolp e))
            (equal (f-nand5 a b c d e) (b-nand5 a b c d e)))
   (implies (and (boolp a) (boolp b) (boolp c) (boolp d) (boolp e)
                 (boolp g))
            (equal (f-nand6 a b c d e g) (b-nand6 a b c d e g)))
   (implies (and (boolp a) (boolp b) (boolp c) (boolp d) (boolp e)
                 (boolp g) (boolp h) (boolp i))
            (equal (f-nand8 a b c d e g h i) (b-nand8 a b c d e g h i)))
   (implies (and (boolp a) (boolp b)) (equal (f-or a b) (b-or a b)))
   (implies (and (boolp a) (boolp b) (boolp c))
            (equal (f-or3 a b c) (b-or3 a b c)))
   (implies (and (boolp a) (boolp b) (boolp c) (boolp d))
            (equal (f-or4 a b c d) (b-or4 a b c d)))
   (implies (and (boolp a) (boolp b)) (equal (f-xor a b) (b-xor a b)))
   (implies (and (boolp a) (boolp b) (boolp c))
            (equal (f-xor3 a b c) (b-xor3 a b c)))
   (implies (and (boolp a) (boolp b)) (equal (f-equv a b) (b-equv a b)))
   (implies (and (boolp a) (boolp b) (boolp c))
            (equal (f-equv3 a b c) (b-equv3 a b c)))
   (implies (and (boolp a) (boolp b)) (equal (f-and a b) (b-and a b)))
   (implies (and (boolp a) (boolp b) (boolp c))
            (equal (f-and3 a b c) (b-and3 a b c)))
   (implies (and (boolp a) (boolp b) (boolp c) (boolp d))
            (equal (f-and4 a b c d) (b-and4 a b c d)))
   (implies (and (boolp a) (boolp b)) (equal (f-nor a b) (b-nor a b)))
   (implies (and (boolp a) (boolp b) (boolp c))
            (equal (f-nor3 a b c) (b-nor3 a b c)))
   (implies (and (boolp a) (boolp b) (boolp c) (boolp d))
            (equal (f-nor4 a b c d) (b-nor4 a b c d)))
   (implies (and (boolp a) (boolp b) (boolp c) (boolp d) (boolp e))
            (equal (f-nor5 a b c d e) (b-nor5 a b c d e)))
   (implies (and (boolp a) (boolp b) (boolp c) (boolp d) (boolp e)
                 (boolp g))
            (equal (f-nor6 a b c d e g) (b-nor6 a b c d e g)))
   (implies (and (boolp a) (boolp b) (boolp c) (boolp d) (boolp e)
                 (boolp g) (boolp h) (boolp i))
            (equal (f-nor8 a b c d e g h i) (b-nor8 a b c d e g h i)))
   (implies (and (boolp c) (boolp a) (boolp b))
            (equal (f-if c a b) (b-if c a b))))
  ;;Hint
  ((enable boolp)))


;;;  FV-OR
```

```
(defn fv-or (a b)
  (if (nlistp a)
      nil
    (cons (f-or (car a) (car b))
          (fv-or (cdr a) (cdr b)))))


(disable fv-or)


(prove-lemma properp-length-fv-or (rewrite)
  (and (properp (fv-or a b))
       (equal (length (fv-or a b))
              (length a)))
  ;;Hint
  ((enable properp fv-or length)))


(prove-lemma fv-or=v-or (rewrite)
  (implies
   (and (bvp a)
        (bvp b)
        (equal (length a) (length b)))
   (equal (fv-or a b)
          (v-or a b)))
  ;;Hint
  ((enable bvp fv-or v-or)))



;;;  FV-XOR


(defn fv-xor (a b)
  (if (nlistp a)
      nil
    (cons (f-xor (car a) (car b))
          (fv-xor (cdr a) (cdr b)))))


(disable fv-xor)


(prove-lemma properp-length-fv-xor (rewrite)
  (and (properp (fv-xor a b))
       (equal (length (fv-xor a b))
              (length a)))
  ;;Hint
  ((enable properp fv-xor length)))
```

```
(prove-lemma fv-xor=v-xor (rewrite)
  (implies
   (and (bvp a)
        (bvp b)
        (equal (length a) (length b)))
   (equal (fv-xor a b)
          (v-xor a b)))
  ;;Hint
  ((enable bvp fv-xor v-xor)))



;;;  FV-IF


(defn fv-if (c a b)
  (if (nlistp a)
      nil
    (cons (f-if c (car a) (car b))
          (fv-if c (cdr a) (cdr b)))))


(disable fv-if)


(prove-lemma length-fv-if (rewrite)
  (equal (length (fv-if c a b))
         (length a))
  ;;Hint
  ((enable length fv-if)))


(prove-lemma properp-fv-if (rewrite)
  (properp (fv-if c a b))
  ;;Hint
  ((enable properp fv-if)))


(prove-lemma v-threefix-fv-if (rewrite)
  (equal (v-threefix (fv-if c a b))
         (fv-if c a b))
  ;;Hint
  ((enable v-threefix fv-if)))
```

```
(prove-lemma fv-if-when-boolp-c (rewrite)
  (implies
   (equal (length a) (length b))
   (and
    (equal (fv-if t a b)
           (v-threefix a))
    (equal (fv-if f a b)
           (v-threefix b))))
  ;;Hint
  ((enable make-list v-threefix length fv-if)))


(prove-lemma fv-if-when-bvp (rewrite)
  (implies
   (and (boolp c)
        (bvp a)
        (bvp b)
        (equal (length a) (length b)))
   (equal (fv-if c a b)
          (if* c a b)))
  ;;Hint
  ((enable if* boolp)))

;;;  Usually disabled to reduce splitting.

(prove-lemma fv-if-rewrite (rewrite)
  (implies
   (equal (length a) (length b))
   (equal (fv-if c a b)
          (if (boolp c)
              (if c (v-threefix a) (v-threefix b))
            (make-list (length a) (x)))))
  ;;Hint
  ((enable length fv-if boolp v-threefix make-list)))


(disable fv-if-rewrite)


(prove-lemma fv-if-v-threefix (rewrite)
  (and
   (equal (fv-if c (v-threefix a) b)
          (fv-if c a b))
   (equal (fv-if c a (v-threefix b))
          (fv-if c a b)))
  ;;Hint
  ((enable fv-if v-threefix)))



;;;   V-WIRE
```

```
(defn v-wire (a b)
  (if (nlistp a)
      nil
    (cons (ft-wire (car a) (car b))
          (v-wire (cdr a) (cdr b)))))


(disable v-wire)


(prove-lemma v-wire-x-x=x (rewrite)
  (equal (v-wire x x)
         (v-fourfix x))
  ;;Hint
  ((enable v-wire v-fourfix)))


(prove-lemma v-wire-make-list-z (rewrite)
  (implies
   (equal (length v) n)
   (and (equal (v-wire (make-list n (z)) v)
               (v-fourfix v))
        (equal (v-wire v (make-list n (z)))
               (v-fourfix v))))
  ;;Hint
  ((induct (nth n v))
   (enable length v-wire make-list v-fourfix fourfix)))


(prove-lemma properp-length-v-wire (rewrite)
  (and (properp (v-wire a b))
       (equal (length (v-wire a b))
              (length a)))
  ;;Hint
  ((enable properp length v-wire)))



(prove-lemma v-wire-make-list-x (rewrite)
  (implies
   (equal n (length x))
   (equal (v-wire x (make-list n (x)))
          (make-list n (x))))
  ;;Hint
  ((enable length v-wire make-list)))


;;;    V-PULLUP
```

```
(defn v-pullup (v)
  (if (nlistp v)
      nil
    (cons (f-pullup (car v))
          (v-pullup (cdr v)))))


(disable v-pullup)


(prove-lemma properp-length-v-pullup (rewrite)
  (and (properp (v-pullup v))
       (equal (length (v-pullup v))
              (length v)))
  ;;Hint
  ((enable properp length v-pullup)))


(prove-lemma v-pullup-bvp (rewrite)
  (implies
   (bvp v)
   (equal (v-pullup v)
          v))
  ;;Hint
  ((enable v-pullup bvp)))


(prove-lemma v-pullup-make-list-z (rewrite)
   (equal (v-pullup (make-list n (z)))
          (make-list n t))
  ;;Hint
  ((enable v-pullup make-list)))



;;;  FV-SHIFT-RIGHT


(defn fv-shift-right (a si)
  (if (nlistp a)
      nil
    (append (v-threefix (cdr a))
            (list (threefix si)))))


(disable fv-shift-right)
```

```
(prove-lemma fv-shift-right=v-shift-right (rewrite)
  (implies
   (and (boolp si)
        (bvp a))
   (equal (fv-shift-right a si)
          (v-shift-right a si)))
  ;;Hint
  ((expand (fv-shift-right a si)
           (v-shift-right a si))))


(prove-lemma properp-length-fv-shift-right (rewrite)
  (and (properp (fv-shift-right a si))
       (equal (length (fv-shift-right a si))
              (length a)))
  ;;Hint
  ((enable fv-shift-right)))


(prove-lemma v-threefix-fv-shift-right (rewrite)
  (equal (v-threefix (fv-shift-right a si))
         (fv-shift-right a si))
  ;;Hint
  ((enable fv-shift-right v-threefix-append)
   (disable append-v-threefix)))



;;;  VFT-BUF - Vector tristate buffer.


(defn vft-buf (c a)
  (if (nlistp a)
      nil
    (cons (ft-buf c (car a))
          (vft-buf c (cdr a)))))


(disable vft-buf)


(prove-lemma properp-length-vft-buf (rewrite)
  (and (properp (vft-buf c a))
       (equal (length (vft-buf c a))
              (length a)))
  ;;Hint
  ((enable properp vft-buf length)))
```

```
(prove-lemma vft-buf-lemmas (rewrite)
  (and
   (equal (vft-buf t a)
          (v-threefix a))
   (equal (vft-buf f a)
          (make-list (length a) (z))))
  ;;Hint
  ((enable vft-buf v-threefix make-list length)))


(prove-lemma vft-buf-rewrite (rewrite)
  (equal (vft-buf c a)
         (if (equal c t)
             (v-threefix a)
           (if (equal c f)
               (make-list (length a) (z))
             (make-list (length a) (x)))))
  ;;Hint
  ((enable vft-buf v-threefix make-list length)))


(disable vft-buf-rewrite)
```

14.30 "dual-eval.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    DUAL-EVAL.EVENTS
;;;
;;;  The definition of the simulator for our HDL.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  The primitive database is created as the Common Lisp object
;;;  COMMON-LISP-PRIMP-DATABASE (see file "primp-database.lisp").  Here, that
;;;  object is used to define the Nqthm constant PRIMP-DATABASE.


(defn primp-database ()
  #.(list 'quote common-lisp-primp-database))

;;;   LOOKUP-MODULE and DELETE-MODULE


(defn lookup-module (name netlist)
  (if (nlistp netlist)
      f
    (if (and (listp (car netlist))
             (equal (caar netlist) name))
        (car netlist)
      (lookup-module name (cdr netlist)))))


(disable lookup-module)


(defn delete-module (name netlist)
  (if (nlistp netlist)
      netlist
    (if (and (listp (car netlist))
             (equal (caar netlist) name))
        (cdr netlist)
      (cons (car netlist) (delete-module name (cdr netlist))))))


(disable delete-module)

;;;  Needed to prove admissibility of DUAL-EVAL.
```

```
(prove-lemma delete-module-lookup-module-count (rewrite)
  (implies
   (lookup-module name netlist)
   (lessp (count (delete-module name netlist))
          (count netlist)))
  ;;Hint
  ((enable lookup-module delete-module)))

;;;   PAIRSTATES and COLLECT-STATES -- PAIRSTATES and COLLECT-STATES are
;;;   written in such a way that if the STATENAMES component of a module
;;;   definition is a single occurrence name, then the new state associated
;;;   with that occurrence is returned as the new state of the module during
;;;   DUAL-EVAL simulation.


(defn pairstates (statenames state)
  (if (or (listp statenames)
          (equal statenames nil))
      (pairlist statenames state)
    (list (cons statenames state))))


(defn collect-states (statenames state-bindings)
  (if (or (listp statenames)
          (equal statenames nil))
      (collect-value statenames state-bindings)
    (value statenames state-bindings)))


;;;   Netlist destructors


(defn module-name        (module) (car module))

(defn module-inputs      (module) (cadr module))

(defn module-outputs     (module) (caddr module))

(defn module-occurrences (module) (cadddr module))

(defn module-statenames  (module) (caddddr module))

(defn module-annotation  (module) (cadddddr module))


(defn occ-name       (occurrence) (car occurrence))

(defn occ-outputs    (occurrence) (cadr occurrence))

(defn occ-function   (occurrence) (caddr occurrence))

(defn occ-inputs     (occurrence) (cadddr occurrence))
```

```
(defn occ-annotation (occurrence) (caddddr occurrence))




(defn primp (fn)
  (lookup-module fn (primp-database)))


(disable primp)

;;;   This lemma avoids having to open up PRIMP when doing proofs
;;;   about modules whose names are not LITATOMs.


(prove-lemma not-litatom==>not-primp (rewrite)
  (implies
   (not (litatom fn))
   (not (primp fn)))
  ;;Hint
  ((enable lookup-module primp)))


(defn primp-lookup (fn name)
  (lookup-module name (cdr (primp fn))))


(defn primp2 (fn name)
  (cdr (primp-lookup fn name)))


(defn dual-apply-value (fn args state)
  (let ((input-names (primp2 fn 'inputs))
        (state-names (primp2 fn 'states))
        (module-outs (primp2 fn 'results)))
    (let ((alist (append (pairlist input-names args)
                         (pairstates state-names state))))
      (eval$ t module-outs alist))))


(disable dual-apply-value)


(defn dual-apply-state (fn args state)
  (let ((input-names (primp2 fn 'inputs))
        (state-names (primp2 fn 'states))
        (new-states  (primp2 fn 'new-states)))
    (let ((alist (append (pairlist input-names args)
                         (pairstates state-names state))))
      (eval$ t new-states alist))))
```

```
(disable dual-apply-state)
```

```
(defn dual-eval (flag x0 x1 x2 netlist)
  (case flag

    ;;  New value of a form.

    (0 (let ((fn x0)
             (args x1)
             (state x2))
         (if (primp fn)
             (dual-apply-value fn args state)

           (let ((module (lookup-module fn netlist)))
             (if (listp module)
                 (let ((inputs  (module-inputs module))
                       (outputs (module-outputs module))
                       (occurrences (module-occurrences module))
                       (statenames (module-statenames module)))
                   (collect-value
                    outputs
                    (dual-eval 1
                               occurrences
                               (pairlist inputs args)
                               (pairstates statenames state)
                               (delete-module fn netlist))))
               f)))))

    ;;  Create the new bindings of a body.

    (1 (let ((body x0)
             (bindings x1)
             (state-bindings x2))

         (if (listp body)
             (let ((occurrence (car body)))
               (let ((occ-name (occ-name occurrence))
                     (outputs (occ-outputs occurrence))
                     (fn (occ-function occurrence))
                     (inputs (occ-inputs occurrence)))
                 (dual-eval
                  1
                  (cdr body)
                  (append
                   (pairlist outputs
                             (dual-eval 0
                                        fn
                                        (collect-value inputs bindings)
                                        (value occ-name state-bindings)
                                        netlist))
                   bindings)
                  state-bindings
                  netlist)))
           bindings)))

    ;;  New state of a form.
```

```
  (2 (let ((fn x0)
           (args x1)
           (state x2))
       (if (primp fn)
           (dual-apply-state fn args state)

         (let ((module (lookup-module fn netlist)))
           (if (listp module)
               (let ((inputs  (module-inputs module))
                     (outputs (module-outputs module))
                     (occurrences (module-occurrences module))
                     (statenames (module-statenames module)))
                 (collect-states
                  statenames
                  (dual-eval 3
                             occurrences
                             (dual-eval 1
                                        occurrences
                                        (pairlist inputs args)
                                        (pairstates statenames state)
                                        (delete-module fn netlist))
                             (pairstates statenames state)
                             (delete-module fn netlist))))
             f)))))

  ;;  New state of a body.

  (3 (let ((body x0)
           (bindings x1)
           (state-bindings x2))

       (if (listp body)
           (let ((occurrence (car body)))
             (let ((occ-name (occ-name occurrence))
                   (outputs (occ-outputs occurrence))
                   (fn (occ-function occurrence))
                   (inputs (occ-inputs occurrence)))
               (cons
                (cons occ-name
                      (dual-eval 2
                                 fn
                                 (collect-value inputs bindings)
                                 (value occ-name state-bindings)
                                 netlist))
                (dual-eval 3
                           (cdr body)
                           bindings
                           state-bindings
                           netlist))))
         nil)))

  (otherwise f))

 ((ord-lessp (cons (add1 (count netlist)) (count x0)))))
```

```
(disable dual-eval)



;;;  This lemma forces DUAL-EVAL with flag 0 to open when we are exploring the
;;;  top-level hierarchical module.


(prove-lemma expand-top-level-dual-eval-0-calls (rewrite)
  (let ((module (lookup-module fn netlist)))
    (implies
     (and (not (primp fn))
          (listp module))
     (equal (dual-eval 0 fn args state netlist)
            (let ((inputs  (module-inputs module))
                  (outputs (module-outputs module))
                  (occurrences (module-occurrences module))
                  (statenames (module-statenames module)))
              (collect-value
               outputs
               (dual-eval 1
                          occurrences
                          (pairlist inputs args)
                          (pairstates statenames state)
                          (delete-module fn netlist)))))))
  ;;Hint
  ((expand (dual-eval 0 fn args state netlist))))

;;;  This lemma forces DUAL-EVAL with flag 2 to open when we are exploring the
;;;  top-level hierarchical module.
```

```
(prove-lemma expand-top-level-dual-eval-2-calls (rewrite)
  (let ((module (lookup-module fn netlist)))
    (implies
     (and (not (primp fn))
          (listp module))
     (equal (dual-eval 2 fn args state netlist)
            (let ((inputs  (module-inputs module))
                  (outputs (module-outputs module))
                  (occurrences (module-occurrences module))
                  (statenames (module-statenames module)))
              (collect-states
               statenames
               (dual-eval 3
                          occurrences
                          (dual-eval 1
                                     occurrences
                                     (pairlist inputs args)
                                     (pairstates statenames state)
                                     (delete-module fn netlist))
                          (pairstates statenames state)
                          (delete-module fn netlist)))))))
  ;;Hint
  ((expand (dual-eval 2 fn args state netlist))))

;;;  Open DUAL-EVAL to evaluate a body for value.
```

```
(prove-lemma open-dual-eval-with-flag-1 (rewrite)
  (and

   (implies
    (nlistp body)
    (equal (dual-eval 1 body bindings state-bindings netlist)
           bindings))

   (implies
    (listp body)
    (equal (dual-eval 1 body bindings state-bindings netlist)
           (let ((occurrence (car body)))
             (let ((occ-name (occ-name occurrence))
                   (outputs (occ-outputs occurrence))
                   (fn (occ-function occurrence))
                   (inputs (occ-inputs occurrence)))
               (dual-eval
                1
                (cdr body)
                (append
                 (pairlist outputs
                           (dual-eval 0
                                      fn
                                      (collect-value inputs bindings)
                                      (value occ-name state-bindings)
                                      netlist))
                 bindings)
                state-bindings
                netlist))))))
  ;;Hint
  ((enable dual-eval)))

;;;  Open DUAL-EVAL to evaluate a body for state.
```

```
(prove-lemma open-dual-eval-with-flag-3 (rewrite)
  (and

   (implies
    (nlistp body)
    (equal (dual-eval 3 body bindings state-bindings netlist)
           nil))

   (implies
    (listp body)
    (equal (dual-eval 3 body bindings state-bindings netlist)
           (let ((occurrence (car body)))
             (let ((occ-name (occ-name occurrence))
                   (outputs (occ-outputs occurrence))
                   (fn (occ-function occurrence))
                   (inputs (occ-inputs occurrence)))
               (cons
                (cons occ-name
                      (dual-eval 2
                                 fn
                                 (collect-value inputs bindings)
                                 (value occ-name state-bindings)
                                 netlist))
                (dual-eval 3
                           (cdr body)
                           bindings
                           state-bindings
                           netlist)))))))
  ;;Hint
  ((enable dual-eval)))

;;;   Some basic properties of DUAL-EVAL.


(prove-lemma properp-dual-eval-0 (rewrite)
  (implies
   (and (not (primp fn))
        (listp (lookup-module fn netlist)))
   (properp (dual-eval 0 fn args state netlist)))
  ;;Hint
  ((expand (dual-eval 0 fn args state netlist))))


(prove-lemma length-dual-eval-0 (rewrite)
  (implies
   (and (not (primp fn))
        (listp (lookup-module fn netlist)))
   (equal (length (dual-eval 0 fn args state netlist))
          (length (module-outputs (lookup-module fn netlist)))))
  ;;Hint
  ((expand (dual-eval 0 fn args state netlist))))
```

```
(prove-lemma properp-dual-eval-2 (rewrite)
  (implies
   (and (not (primp fn))
        (listp (lookup-module fn netlist))
        (properp (module-statenames (lookup-module fn netlist))))
   (properp (dual-eval 2 fn args state netlist)))
  ;;Hint
  ((expand (dual-eval 2 fn args state netlist))))


(prove-lemma length-dual-eval-2 (rewrite)
  (implies
   (and (not (primp fn))
        (listp (lookup-module fn netlist))
        (properp (module-statenames (lookup-module fn netlist))))
   (equal (length (dual-eval 2 fn args state netlist))
          (length (module-statenames (lookup-module fn netlist)))))
  ;;Hint
  ((expand (dual-eval 2 fn args state netlist))))



;;;  DUAL-EVAL-BODY-BINDINGS is a shorthand way of generating the binding
;;;  lists when doing proofs about the bodies of modules.


(defn dual-eval-body-bindings (n body bindings state-bindings netlist)
  (if (zerop n)
      bindings
    (let ((occurrence (car body)))
      (let ((occ-name (occ-name occurrence))
            (outputs (occ-outputs occurrence))
            (fn (occ-function occurrence))
            (inputs (occ-inputs occurrence)))
        (dual-eval-body-bindings
         (sub1 n)
         (cdr body)
         (append (pairlist outputs
                           (dual-eval 0
                                      fn
                                      (collect-value inputs bindings)
                                      (value occ-name state-bindings)
                                      netlist))
                 bindings)
         state-bindings
         netlist)))))


(disable dual-eval-body-bindings)
```

```
(prove-lemma open-dual-eval-body-bindings (rewrite)
  (and
   (implies
    (zerop n)
    (equal (dual-eval-body-bindings n body bindings state-bindings netlist)
           bindings))
   (implies
    (not (zerop n))
    (equal (dual-eval-body-bindings n body bindings state-bindings netlist)
           (let ((occurrence (car body)))
      (let ((occ-name (occ-name occurrence))
            (outputs (occ-outputs occurrence))
            (fn (occ-function occurrence))
            (inputs (occ-inputs occurrence)))
        (dual-eval-body-bindings
         (sub1 n)
         (cdr body)
         (append (pairlist outputs
                           (dual-eval 0
                                      fn
                                      (collect-value inputs bindings)
                                      (value occ-name state-bindings)
                                      netlist))
                 bindings)
         state-bindings
         netlist))))))
  ;;Hint
  ((enable dual-eval-body-bindings)))



;;;   SIMULATE
;;;
;;;   Runs DUAL-EVAL over time.


(defn simulate (fn inputs state netlist)
  (if (nlistp inputs)
      nil
    (let ((value (dual-eval 0 fn (car inputs) state netlist))
          (new-state (dual-eval 2 fn (car inputs) state netlist)))
      (cons (list value new-state)
            (simulate fn (cdr inputs) new-state netlist)))))

;;;  SIMULATE-DUAL-EVAL-2
;;;
;;;  Returns the final state after a number of simulation steps.
```

```
(defn simulate-dual-eval-2 (module inputs state netlist n)
  (if (zerop n)
      state
    (simulate-dual-eval-2
     module
     (cdr inputs)
     (dual-eval 2 module (car inputs) state netlist)
     netlist
     (sub1 n))))


(disable simulate-dual-eval-2)


(prove-lemma open-simulate-dual-eval-2 (rewrite)
  (and
   (implies
    (zerop n)
    (equal (simulate-dual-eval-2 module inputs state netlist n)
           state))
   (implies
    (not (zerop n))
    (equal (simulate-dual-eval-2 module inputs state netlist n)
           (simulate-dual-eval-2
            module
            (cdr inputs)
            (dual-eval 2 module (car inputs) state netlist)
            netlist
            (sub1 n)))))
  ;;Hint
  ((enable simulate-dual-eval-2)))
```

14.31 "predicate-help.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    PREDICATE-HELP.EVENTS
;;;
;;;  This file contains definitions that are used in file "predicate.events".
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;; ~~~~~~~~~~~~
;;;  Arithmetic
;;; ~~~~~~~~~~~~


(defn min (x y)
  (if (lessp y x)
      y
      x))


(disable min)


;;; ~~~~~~
;;;  Sets
;;; ~~~~~~


(defn insert (e x)
  (if (member e x)
      x
      (cons e x)))


(disable insert)


(defn set-equal (x y)
  (and (subset x y)
       (subset y x)))


(disable set-equal)
```

```
(defn set-diff (x y)
  (if (listp x)
      (let ((e (car x))
            (r (cdr x)))
        (if (member e y)
            (set-diff r y)
            (cons e (set-diff r y))))
      nil))

(disable set-diff)

(defn intersection (x y)
  (if (listp x)
      (let ((e (car x))
            (r (cdr x)))
        (if (member e y)
            (cons e (intersection r y))
            (intersection r y)))
      nil))

(disable intersection)


;;; ~~~~~~~
;;;  Lists
;;; ~~~~~~~

(defn last-cdr (x)
  (if (listp x)
      (last-cdr (cdr x))
      x))

(disable last-cdr)

(defn list-duplicates (x duplicates)
  (if (listp x)
      (let ((e (car x))
            (r (cdr x)))
        (list-duplicates r (if (member e r)
                               (insert e duplicates)
                               duplicates)))
      duplicates))

(disable list-duplicates)
```

```
(defn remove-duplicates (x)
  (if (listp x)
      (let ((e (car x))
            (r (cdr x)))
        (if (member e r)
            (remove-duplicates r)
            (cons e (remove-duplicates r))))
      x))

(disable remove-duplicates)

(defn append-list (x)
  (if (listp x)
      (append (car x)
              (append-list (cdr x)))
      nil))

(disable append-list)

(defn flatten-list (x)
  (remove-duplicates (append-list x)))

(disable flatten-list)

(defn is-head (x y)
  (if (and (listp x) (listp y))
      (and (equal (car x) (car y))
           (is-head (cdr x) (cdr y)))
      (nlistp x)))

(disable is-head)

(defn listify (x)
  (if (listp x)
      (cons (list (car x))
            (listify (cdr x)))
      nil))
```

```
(disable listify)


;;; ~~~~~~~~
;;;  Alists
;;; ~~~~~~~~


(defn alistp (x)
  (if (listp x)
      (and (listp (car x))
           (alistp (cdr x)))
      (equal x nil)))


(disable alistp)


(defn boundp (key alist)
  (listp (lookup-module key alist)))

;; (defn value (key alist) ...) is in file "dual-eval.events".


(defn alist-entry (key alist)
  (lookup-module key alist))


(disable alist-entry)


(defn unbind (key alist)
  (delete-module key alist))


(defn set-value (key value alist)
  (let ((entry (alist-entry key alist)))
    (cond ((nlistp entry)
           (cons (cons key value) alist))
          ((equal value (cdr entry))
           alist)
          (T (cons (cons key value)
                   (unbind key alist))))))


(disable set-value)
```

```
(defn union-values (keys alist)
  (if (listp keys)
      (union (value (car keys) alist)
             (union-values (cdr keys) alist))
      nil))


(disable union-values)


(defn list-union-values (list-of-args alist)
  ;; list-of-args is a list of lists
  (if (listp list-of-args)
      (cons (union-values (car list-of-args) alist)
            (list-union-values (cdr list-of-args) alist))
      nil))


(disable list-union-values)



(defn unbound-keys (keys alist)
  (if (listp keys)
      (let ((key1 (car keys))
            (rslt (unbound-keys (cdr keys) alist)))
        (if (boundp key1 alist)
            rslt
            (insert key1 rslt)))
      nil))


(disable unbound-keys)


(defn unbind-list (keys alist)
  (if (listp keys)
      (unbind (car keys)
              (unbind-list (cdr keys) alist))
      alist))


(prove-lemma lookup-module-in-delete-module (rewrite)
  (implies (and (not (equal k1 k2))
                (lookup-module k1 m))
           (lookup-module k1 (delete-module k2 m)))
  ((enable delete-module lookup-module)))


(prove-lemma lookup-module-in-unbind-list (rewrite)
  (implies (and (not (member k u))
                (lookup-module k m))
           (lookup-module k (unbind-list u m))))
```

```
(disable lookup-module-in-delete-module)



(prove-lemma lessp-unbind-list-count (rewrite)
  (implies (and (not (member k u))
                (boundp k m))
           (lessp (count (unbind-list (cons k u) m))
                  (count (unbind-list u m)))))


(disable boundp)

(disable unbind)

(disable unbind-list)

(disable lookup-module-in-unbind-list)


;;; ~~~~~~~~~
;;;  Netlist
;;; ~~~~~~~~~


(defn m-states-list (m-states)
  (if (or (listp m-states)
          (equal m-states nil))
      m-states
    (list m-states)))


(disable m-states-list)




;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Sorting Names
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
(defn list-lessp (x y)
  (if (listp x)
      (if (listp y)
          (if (lessp (car x) (car y))
              t
              (if (equal (car x) (car y))
                  (list-lessp (cdr x) (cdr y))
                  f))
          f)
      (listp y)))
```

```
(disable list-lessp)
```

```
(defn token-lessp (x y)
  (cond ((and (litatom x)
              (litatom y))
         (list-lessp (unpack x) (unpack y)))
        ((and (litatom x)
              (indexp y))
         t)
        ((and (indexp x)
              (litatom y)) f)
        ((and (indexp x)
              (indexp y))
         (if (equal (i-name x) (i-name y))
             (lessp (i-num x) (i-num y))
             (list-lessp (unpack (i-name x))
                         (unpack (i-name y)))))
        (t f)))
```

```
(disable token-lessp)
```

```
(defn min-token-help (seed list)
  (if (listp list)
      (if (token-lessp seed (car list))
          (min-token-help seed (cdr list))
          (min-token-help (car list) (cdr list)))
      seed))
```

```
(disable min-token-help)
```

```
(defn min-token (list)
  (if (listp list)
      (if (listp (cdr list))
          (min-token-help (car list) (cdr list))
          (car list))
      0))


(disable min-token)



(prove-lemma min-token-help-lemma nil
  (or (equal (min-token-help seed list) seed)
      (member (min-token-help seed list) list))
  ((enable min-token-help)))


(prove-lemma min-token-lemma (rewrite)
  (implies (listp lst)
           (member (min-token lst) lst))
  ((use (min-token-help-lemma (seed (car lst)) (list (cdr lst))))
   (enable min-token)))



(prove-lemma delete*-count (rewrite)
  (lessp (count (delete* e x))
         (count (cons e x)))
  ((enable delete*)))


(prove-lemma delete*-lemma (rewrite)
  (implies (member e lst)
           (lessp (count (delete* e lst))
                  (count lst)))
  ((enable delete*)))


(defn list-sort (list)
  (if (listp list)
      (cons (min-token list)
            (list-sort (delete* (min-token list) list)))
      nil))


(disable list-sort)
```

```
(defn list-list-sort (list)
  (if (listp list)
      (cons (list-sort (car list))
            (list-list-sort (cdr list)))
      nil))


(disable list-list-sort)




;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Errors
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(add-shell net-error nil net-errorp
           ((error-label (none-of) false)
            (error-body (none-of) true)))

;; Redefine *1*report-error to make errors break into Lisp.


(defn report-error (err)
  err)


(disable report-error)


(defn pred-error (label body)
  (report-error (net-error label body)))


(disable pred-error)


(defn error-entry (lst label)
  (if (listp lst)
      (let ((e (car lst))
            (r (cdr lst)))
        (if (net-errorp e)
            (if (equal (error-label e) label)
                e
                (error-entry r label))
            (error-entry r label)))
      F))


(disable error-entry)
```

```
(defn net-errors (lst)
  (if (listp lst)
      (let ((e (car lst))
            (r (cdr lst)))
        (if (net-errorp e)
            (cons e (net-errors r))
            (net-errors r)))
      nil))
```

```
(disable net-errors)
```

```
(defn collect-net-errors (lst)
  (if (listp lst)
      (append (net-errors (car lst))
              (collect-net-errors (cdr lst)))
      nil))
```

```
(disable collect-net-errors)
```

```
(defn err-and (lst label)
  (let ((r (net-errors lst)))
    (if (nlistp r)
        T
        (pred-error label r))))
```

```
(disable err-and)
```

```
(defn T-or-err (p label body)
  (if (and p (not (net-errorp p)))
      t
      (pred-error label body)))
```

```
(disable T-or-err)
```

```
(defn nlistp-or-err (x label)
  (if (nlistp x)
      T
      (pred-error label x)))
```

```
(disable nlistp-or-err)
```

```
(defn nil-or-err (x label)
  (if (equal x nil)
      T
      (pred-error label x)))


(disable nil-or-err)


(defn subset-or-err (x y label)
  (let ((r (set-diff x y)))
    (if (nlistp r)
        T
        (pred-error label r))))


(disable subset-or-err)


(defn no-duplicates-or-err (x label)
  (let ((r (list-duplicates x nil)))
    (if (nlistp r)
        T
        (pred-error label r))))


(disable no-duplicates-or-err)


(defn disjoint-or-err (x y label)
  (let ((r (intersection x y)))
    (if (nlistp r)
        T
        (pred-error label r))))


(disable disjoint-or-err)


(defn all-bound-or-err (keys alist label)
  (let ((r (unbound-keys keys alist)))
    (if (nlistp r)
        T
        (pred-error label r))))


(disable all-bound-or-err)


(defn label-error (x label)
  (if (net-errorp x)
      (pred-error label x)
      x))
```

```
(disable label-error)


(defn list-collect-value (list-of-args alist)
  ; list-of-args is a list of lists
  (if (nlistp list-of-args)
      nil
      (cons (flatten-list
              (collect-value (car list-of-args) alist))
            (list-collect-value (cdr list-of-args) alist))))


(disable list-collect-value)
```

14.32 "predicate-simple.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    PREDICATE-SIMPLE.EVENTS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;  Syntax Requirements for Hardware Description Language
;;;
;;;  The predicate TOP-LEVEL-PREDICATE-SIMPLE is a simpler version
;;;  of the circuit predicate in file "predicate.events".  These
;;;  predicates only return a symbol indicating what main property
;;;  was violated.  The circuit recognizer predicates in file
;;;  "predicate.events" return more useful information, but the
;;;  functions there are much more complex.
;;;
;;;  We don't use these circuit predicates during our proofs; however,
;;;  once we have a netlist we check its well-formedness using these
;;;  predicates.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn name-simple-okp (name)
  (or (litatom name)
      (and (indexp name)
           (litatom (i-name name)))))


(disable name-simple-okp)


(defn name-list-simple-okp (name-list)
  (if (nlistp name-list)
      (equal name-list nil)
    (let ((name (car name-list))
          (rest (cdr name-list)))
      (and (name-simple-okp name)
           (not (member name rest))
           (name-list-simple-okp rest)))))


(disable name-list-simple-okp)
```

```
(defn occ-syntax-simple-okp (occ netlist) ;; occ is a single occurrence

  (let ((o-name (occ-name       occ))
        (o-outs (occ-outputs    occ))
        (o-fn   (occ-function   occ))
        (o-ins  (occ-inputs     occ))
        (o-anno (occ-annotation occ)))

    (and (or (equal 4 (length occ))
             (and (equal 5 (length occ))
                  (alistp o-anno)))
         (properp occ)

         (name-simple-okp o-name)
         (name-list-simple-okp o-outs)
         (properp o-ins)

         ; We cannot check NAME-LIST-SIMPLE-OKP for O-INS because of
         ; wiring fanouts.  All O-INS names are checked
         ; elsewhere because they must ultimately be a subset
         ; of signals, which are checked either as module inputs
         ; or as occurrence outputs.

         (if (primp o-fn)

             (and (equal (length (primp2 o-fn 'inputs))
                         (length o-ins))
                  (equal (length (primp2 o-fn 'outputs))
                         (length o-outs)))

           (let ((module (lookup-module o-fn netlist)))
             (let ((lm-ins  (module-inputs  module))
                   (lm-outs (module-outputs module)))

               (and (listp module)
                    (equal (length lm-ins)  (length o-ins))
                    (equal (length lm-outs) (length o-outs)))))))))
```

```
(disable occ-syntax-simple-okp)


;;;  In OCC-BODY-SYNTAX-SIMPLE-OKP, the parameters are as follows:
;;;    body - the list of occurrences from a module (or a tail of it).
;;;    signals - a list of defined signals, consisting of module inputs and
;;;              outputs of occurrences previous to body.
;;;    non-depends - a list of signals that were not defined when they were
;;;                  used as inputs of ocurrences previous to body.  The
;;;                  outputs of those occurrences should not have depended on
;;;                  the undefined inputs, but that check is made elsewhere
;;;                  (in function OUT-DEPENDS-SIMPLE).
;;;    occ-names - a list of the names of occurrences previous to body.
;;;    outputs - a list of the outputs of the module.
;;;    states - a list of the state names of the module.
;;;    netlist - the netlist


(defn occ-body-syntax-simple-okp (body signals non-depends occ-names
                                       outputs states netlist)
  (if (nlistp body)

      (and (equal body nil)

           (subset outputs signals)
           (subset non-depends signals)
           (subset states occ-names)

           (not (duplicates? signals))
           (not (duplicates? occ-names)))

    (let ((occ (car body)))
      (let ((o-name (occ-name    occ))
            (o-outs (occ-outputs occ))
            (o-ins  (occ-inputs  occ)))

        (if (occ-syntax-simple-okp occ netlist)

            (occ-body-syntax-simple-okp (cdr body)
                                        (append o-outs signals)
                                        (append (set-diff o-ins signals)
                                                non-depends)
                                        (cons o-name occ-names)
                                        outputs states
                                        netlist)
          f)))))


(disable occ-body-syntax-simple-okp)
```

```
(defn module-syntax-simple-okp (netlist)

  (let ((module (car netlist))
        (net-rest (cdr netlist)))

    (let ((m-name   (module-name        module))
          (m-ins    (module-inputs      module))
          (m-outs   (module-outputs     module))
          (m-occs   (module-occurrences module))
          (m-states (module-statenames  module))
          (m-anno   (module-annotation  module)))

      (let ((m-states-list (m-states-list m-states)))

        (and (or (equal (length module) 5)
                 (and (equal (length module) 6)
                      (alistp m-anno)))
             (properp module)

             (name-simple-okp m-name)
             (not (lookup-module m-name net-rest))
             (not (primp m-name))

             (name-list-simple-okp m-ins)
             (name-list-simple-okp m-outs)
             (name-list-simple-okp m-states-list)

             (disjoint m-ins m-outs)

             (listp m-occs)

             (occ-body-syntax-simple-okp m-occs m-ins nil nil m-outs
                                         m-states-list net-rest))))))


(disable module-syntax-simple-okp)


(defn netlist-syntax-simple-okp (netlist)
  (if (nlistp netlist)

      (equal netlist nil)

    (if (module-syntax-simple-okp netlist)
        (netlist-syntax-simple-okp (cdr netlist))
      f)))
```

```
(disable netlist-syntax-simple-okp)


;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Output Dependencies
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  In OUT-DEPENDS-SIMPLE, module-database can be a table of
;;;  previously computed output dependencies (see function
;;;  dependency-table).  Without this table, OUT-DEPENDS-SIMPLE
;;;  recomputes output dependencies for a module every time it is
;;;  used.  With the table, OUT-DEPENDS-SIMPLE merely looks up (and
;;;  fixes) the precomputed value.  (Fixing the value means replacing
;;;  local names with the dependencies provided as an argument.)
```

```
(defn out-depends-simple (flg x0 x1 netlist module-database)
  (case
    flg
    (0 (let ((fn x0)     ; name of a primitive or a module in netlist
             (args x1))  ; list of dependency lists for fn inputs

         (cond ((primp fn)
                (list-collect-value
                 (primp2 fn 'out-depends)
                 (pairlist (primp2 fn 'inputs) args)))

               ((lookup-module 'out-depends
                               (cdr (lookup-module fn module-database)))
                (list-collect-value
                 (cdr (lookup-module
                       'out-depends
                       (cdr (lookup-module fn module-database))))
                 (pairlist (module-inputs (lookup-module fn netlist))
                           args)))

               (t (let ((module (lookup-module fn netlist)))
                    (if (listp module)
                        (let ((m-ins  (module-inputs      module))
                              (m-outs (module-outputs     module))
                              (m-occs (module-occurrences module)))

                          (let ((alist-depends
                                 (out-depends-simple 1
                                                     m-occs
                                                     (pairlist m-ins args)
                                                     (delete-module fn netlist)
                                                     module-database)))

                            (if (and alist-depends
                                     (subset m-outs
                                             (strip-cars alist-depends)))
                                (collect-value m-outs alist-depends)
                              f)))
                      f))))))

    (1 (let ((body x0)      ; list of occurrences from a module (or a tail of
                            ; it)
             (bindings x1)) ; alist binding each signal to its INPUT
                            ; dependencies

         (if (nlistp body)
             bindings

             (let ((occ (car body)))
               (let ((o-outs (occ-outputs  occ))
                     (o-fn   (occ-function occ))
                     (o-ins  (occ-inputs   occ)))

                 (let ((local-depend-list
```

```
                       (out-depends-simple 0 o-fn
                                   (listify o-ins)
                                   netlist module-database)))
                  (if (and (subset (flatten-list local-depend-list)
                                  (strip-cars bindings))
                          local-depend-list)

                      (out-depends-simple 1 (cdr body)
                                  (append (pairlist o-outs
                                                   (list-collect-value
                                                    local-depend-list
                                                    bindings))
                                          bindings)
                                  netlist module-database)
                      f)))))))

    (otherwise f))

  ((ord-lessp (cons (add1 (count netlist)) (count x0)))))


(disable out-depends-simple)




(defn simple-dependency-table (netlist)

  (if (nlistp netlist)
      nil

    (let ((module (car netlist))
          (net-rest (cdr netlist)))

      (let ((t2 (simple-dependency-table net-rest))
            (m-name (module-name    module))
            (m-ins  (module-inputs  module)))

        (if t2

            (let ((dependencies (out-depends-simple 0
                                              m-name
                                              (listify m-ins)
                                              netlist t2)))

              (if dependencies
                  (cons (list m-name
                              (cons 'out-depends dependencies))
                        t2)
                f))
          f)))))
```

```
(disable simple-dependency-table)


;;;  Function output-dependencies-simple is a utility function (not part of the
;;;  predicate).


(defn output-dependencies-simple (module-name args netlist)

  (if (or (and (primp module-name)
               (equal (length args)
                      (length (primp2 module-name 'inputs))))

          (and (lookup-module module-name netlist)
               (equal (length args)
                      (length (module-inputs
                               (lookup-module module-name
                                              netlist))))))
      (list-list-sort
       (out-depends-simple 0 module-name args netlist
                    (simple-dependency-table netlist)))

    f))


(disable output-dependencies-simple)




;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Input/Output Types
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  Below are functions that check the types of module inputs and outputs.


(defn type-value-simple (n alist default)
  (let ((r (lookup-module n alist)))
    (if (nlistp r)
        (if default 'free (list n))
      (if (nlistp (cdr r))
          (cdr r)
        (if (equal (cadr r) n)
            (if default 'free (cdr r))
          (type-value-simple (cadr r)
                             (delete-module n alist)
                             default))))))


(disable type-value-simple)
```

```
(defn collect-types (names alist default)
  (if (nlistp names)
      nil
    (cons (type-value-simple (car names) alist default)
          (collect-types (cdr names) alist default))))
```

```
(disable collect-types)
```

```
(defn add-new-type (new-type-entry known-types)
  (let ((name (car new-type-entry))
        (new-type (cdr new-type-entry)))
    (let ((old-type-entry (lookup-module name known-types)))
      (let ((old-type (cdr old-type-entry)))
        (cond ((equal new-type 'free) known-types)
              ((nlistp old-type-entry) (cons new-type-entry known-types))
              ((equal new-type old-type) known-types)
              ((listp new-type)
               (cons (cons (car new-type) old-type)
                     known-types))
              ((listp old-type)
               (let ((r (add-new-type (cons (car old-type) new-type)
                                      (delete-module name known-types))))
                 (if r (cons new-type-entry r) f)))
              (t f))))))
```

```
(disable add-new-type)
```

```
(defn update-known-types (new-types known-types)
  (if (or (nlistp new-types)
          (falsep known-types))
      known-types
    (update-known-types (cdr new-types)
                        (add-new-type (car new-types) known-types))))
```

```
(disable update-known-types)
```

```
(defn fix-dependent-types (types alist)
  (if (nlistp types)
      nil
    (if (listp (car types))
        (cons (value (caar types) alist)
              (fix-dependent-types (cdr types) alist))
      (cons (car types)
            (fix-dependent-types (cdr types) alist)))))
```

```
(disable fix-dependent-types)
```

```
(defn io-types-simple (flg x0 x1 netlist type-table)
  (case
    flg
    (0 (let ((fn x0)
             (arg-types x1))

         (cond ((primp fn)
                (cons (primp2 fn 'input-types)
                      (fix-dependent-types
                       (primp2 fn 'output-types)
                       (pairlist (primp2 fn 'inputs) arg-types))))

               ((and (lookup-module 'input-types
                                    (cdr (lookup-module fn type-table)))
                     (lookup-module 'output-types
                                    (cdr (lookup-module fn type-table))))
                (cons (cdr (lookup-module
                            'input-types
                            (cdr (lookup-module fn type-table))))
                      (fix-dependent-types
                       (cdr (lookup-module
                             'output-types
                             (cdr (lookup-module fn type-table))))
                       (pairlist (module-inputs (lookup-module fn netlist))
                                 arg-types))))

               (t (let ((module (lookup-module fn netlist)))
                    (if (listp module)
                        (let ((m-ins  (module-inputs      module))
                              (m-outs (module-outputs     module))
                              (m-occs (module-occurrences module)))

                          (let ((o-types
                                 (io-types-simple 1 m-occs nil
                                                  (delete-module fn netlist)
                                                  type-table)))

                            (if o-types
                                (cons (collect-types m-ins o-types t)
                                      (fix-dependent-types
                                       (collect-types m-outs o-types f)
                                       (pairlist m-ins arg-types)))
                              f)))

                      f))))))

    (1 (let ((body x0)
             (known-types x1))

         (if (nlistp body)
             known-types

           (let ((occ (car body)))
             (let ((o-outs (occ-outputs  occ))
```

```
                   (o-fn   (occ-function occ))
                   (o-ins  (occ-inputs   occ)))

               (let ((o-types
                      (io-types-simple 0 o-fn
                                       (collect-types o-ins known-types f)
                                       netlist type-table)))
                 (if (listp o-types)
                     (let ((o-in-types  (car o-types))
                           (o-out-types (cdr o-types)))
                       (let ((new-types (update-known-types
                                         (pairlist o-outs o-out-types)
                                         (update-known-types
                                          (pairlist o-ins o-in-types)
                                          known-types))))
                         (if new-types

                             (io-types-simple 1 (cdr body) new-types
                                              netlist type-table)
                           f)))

                   f)))))))

    (otherwise f))

  ((ord-lessp (cons (add1 (count netlist)) (count x0)))))


(disable io-types-simple)




(defn netlist-type-table-simple (netlist)

  (if (nlistp netlist)
      nil

    (let ((r (netlist-type-table-simple (cdr netlist)))
          (module (car netlist)))

      (if r
          (let ((m-name (module-name    module))
                (m-ins  (module-inputs  module)))
            (let ((io-types-simple
                   (io-types-simple 0 m-name (listify m-ins) netlist r)))

              (if (listp io-types-simple)
                  (cons (list m-name
                              (cons 'input-types (car io-types-simple))
                              (cons 'output-types (cdr io-types-simple)))
                        r)
                f)))
        f))))
```

```
(disable netlist-type-table-simple)
```

```
(defn types-acceptablep (flg x0 netlist acceptable-types checked-modules)
  (case
    flg
    (0 (let ((fn x0))

         (cond ((primp fn)
                (and (subset (primp2 fn 'input-types) acceptable-types)
                     (subset (fix-dependent-types
                              (primp2 fn 'output-types)
                              (pairlist (primp2 fn 'inputs)
                                        (primp2 fn 'input-types)))
                             acceptable-types)))

               ((member fn checked-modules) t)

               (t (let ((module (lookup-module fn netlist)))

                    (if (listp module)
                        (let ((m-occs (module-occurrences module)))

                          (types-acceptablep 1 m-occs
                                             (delete-module fn netlist)
                                             acceptable-types
                                             checked-modules))
                      f))))))

    (1 (let ((body x0))

         (if (nlistp body)
             t

           (let ((occ (car body)))
             (let ((o-fn (occ-function occ)))

               (if (types-acceptablep 0 o-fn netlist
                                      acceptable-types checked-modules)
                   (types-acceptablep 1 (cdr body) netlist
                                      acceptable-types checked-modules)
                 f))))))

    (otherwise f))

  ((ord-lessp (cons (add1 (count netlist)) (count x0)))))
```

```
(disable types-acceptablep)
```

```
(defn netlist-types-acceptable-list (netlist)
  (if (nlistp netlist)
      nil
    (let ((r (netlist-types-acceptable-list (cdr netlist)))
          (module (car netlist)))
      (if r
          (let ((m-name (module-name module)))
            (if (types-acceptablep 0 m-name netlist
                                   '(boolp clk level clk-10 free)
                                   r)
                (cons m-name r)
              f))
        f))))


(disable netlist-types-acceptable-list)




(defn netlist-type-check-simple-okp (netlist)
  (if (netlist-types-acceptable-list netlist)
      (netlist-type-table-simple netlist)
    f))


(disable netlist-type-check-simple-okp)


;;; The following 2 functions are utility functions (not part of the
;;; predicate).


(defn arg-types-match-simplep (actuals formals)
  (if (or (nlistp actuals) (nlistp formals))
      (equal actuals formals)
    (and (or (equal (car actuals) (car formals))
             (equal (car formals) 'free))
         (arg-types-match-simplep (cdr actuals) (cdr formals)))))


(disable arg-types-match-simplep)
```

```
(defn arg-types-simple-okp (fn arg-types netlist)

  (if (not (subset arg-types '(boolp clk level clk-10)))
      f

    (let ((io-types-simple
           (io-types-simple 0 fn arg-types netlist
                            (netlist-type-table-simple netlist))))
      (if (listp io-types-simple)
          (arg-types-match-simplep arg-types (car io-types-simple))
        f))))


(disable arg-types-simple-okp)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    State Types
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  The predicates below check that a state argument for DUAL-EVAL
;;;  is well-formed (the correct type) with respect to the netlist.



(defn make-ram-state (structure width bit-value)
  (if (nlistp structure)
      (ram (make-list width bit-value))
    (cons (make-ram-state (car structure) width bit-value)
          (make-ram-state (cdr structure) width bit-value))))


(disable make-ram-state)
```

```
(defn state-type-requirement-simple (flg x0 x1 netlist type-table)
  (case
    flg
    (0 (let ((fn x0))

         (cond ((primp fn)
                (if (lookup-module 'state-types (cdr (primp fn)))
                    (eval$ t
                           (cdr (lookup-module 'state-types (cdr (primp fn))))
                           nil)
                  T))

               ((lookup-module fn type-table)
                (if (lookup-module 'state-types
                                   (cdr (lookup-module fn type-table)))
                    (cdr (lookup-module 'state-types
                                        (cdr (lookup-module fn type-table))))
                  t))

               (t (let ((module (lookup-module fn netlist)))
                    (if (listp module)
                        (let ((m-occs   (module-occurrences module))
                              (m-states (module-statenames  module)))
                          (let ((m-state-list (m-states-list m-states))
                                (state-types
                                 (state-type-requirement-simple
                                  1 m-occs nil
                                  (delete-module fn netlist)
                                  type-table)))
                            (if (set-equal m-state-list (strip-cars
                                                         state-types))
                                (if (equal m-states nil)
                                    T
                                  (collect-states m-states state-types))
                              f)))
                      f))))))

    (1 (let ((body x0)
             (state-types x1))

         (if (nlistp body)
             state-types

           (let ((occ (car body)))
             (let ((o-name (occ-name     occ))
                   (o-fn   (occ-function occ)))

               (let ((o-type
                      (state-type-requirement-simple 0 o-fn nil
                                                     netlist type-table)))
                 (if (truep o-type)

                     (state-type-requirement-simple 1 (cdr body) state-types
                                                    netlist type-table)
```

```
                   (if o-type
                       (state-type-requirement-simple
                        1 (cdr body)
                        (cons (cons o-name o-type) state-types)
                        netlist type-table)

                     f))))))))

    (otherwise f))

  ((ord-lessp (cons (add1 (count netlist)) (count x0)))))


(disable state-type-requirement-simple)




(defn netlist-state-types-simple (netlist)

  (if (nlistp netlist)
      nil

    (let ((r (netlist-state-types-simple (cdr netlist)))
          (module (car netlist)))

      (if r
          (let ((m-name (module-name module)))
            (let ((state-types (state-type-requirement-simple 0 m-name nil
                                                               netlist r)))

              (if (truep state-types)
                  r
                (if state-types
                    (cons (list m-name
                                (cons 'state-types state-types))
                          r)
                  f))))
        f))))


(disable netlist-state-types-simple)



;;;  The following 2 functions are utility functions (not part of the
;;;  predicate).
```

```
(defn state-simple-okp (state type)
  (cond ((truep type)
         (equal state nil))
        ((equal type 'boolp)
         (boolp state))
        ((ramp type)
         (and (ramp state)
              (state-simple-okp (ram-guts state) (ram-guts type))))
        ((romp type)
         (and (romp state)
              (state-simple-okp (rom-guts state) (rom-guts type))))
        ((stubp type)
         (and (stubp state)
              (state-simple-okp (stub-guts state) (stub-guts type))))
        ((listp type)
         (and (listp state)
              (state-simple-okp (car state) (car type))
              (state-simple-okp (cdr state) (cdr type))))
        ((equal type nil)
         (equal state nil))
        (T F)))


(disable state-simple-okp)



(defn apply-state-simple-okp (fn state netlist)
  (let ((type (state-type-requirement-simple
               0 fn nil netlist (netlist-state-types-simple netlist))))
    (state-simple-okp state type)))


(disable apply-state-simple-okp)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Netlist Loadings and Drives
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  Note:  This assumes each loading is a number, a name, or a list of
;;;  number and names, and each drive value is a number or a name.
```

```
(defn fix-dependent-lds (lds alist)
  (if (nlistp lds)
      lds
    (let ((ld (cond ((or (litatom (car lds)) (indexp (car lds)))
                     (value (car lds) alist))
                    ((nlistp (car lds)) (car lds))
                    (t (fix-dependent-lds (car lds) alist)))))
      (cons ld (fix-dependent-lds (cdr lds) alist)))))


(disable fix-dependent-lds)


(defn parent-synonym-simple (name slist)
  (if (listp (lookup-module name slist))
      (let ((v (cdr (lookup-module name slist))))
        (if (or (litatom v) (indexp v))
            (parent-synonym-simple v (delete-module name slist))
          name))
    name))


(disable parent-synonym-simple)


(defn add-loading-simple (name loading ld-list)
  (if (nlistp (lookup-module name ld-list))
      (cons (cons name (if (listp loading) loading (list loading)))
            ld-list)
    (let ((old-loadings (cdr (lookup-module name ld-list))))
      (cons (cons name
                  (if (listp loading)
                      (append loading old-loadings)
                    (cons loading old-loadings)))
            (delete-module name ld-list)))))


(disable add-loading-simple)


(defn add-loading-simples (names loadings ld-list)
  (if (nlistp names)
      ld-list
    (add-loading-simples (cdr names) (cdr loadings)
                         (add-loading-simple (car names) (car loadings)
                                             ld-list))))


(disable add-loading-simples)
```

```
(defn sum-numbers (lst)
  (cond ((nlistp lst) 0)
        ((numberp (car lst))
         (plus (car lst) (sum-numbers (cdr lst))))
        (t (sum-numbers (cdr lst)))))


(disable sum-numbers)


(defn extract-names-simple (lst)
  (cond ((nlistp lst) nil)
        ((and (or (litatom (car lst)) (indexp (car lst)))
              (not (member (car lst) (cdr lst))))
         (cons (car lst) (extract-names-simple (cdr lst))))
        (t (extract-names-simple (cdr lst)))))


(disable extract-names-simple)


(defn sum-loading (ld)
  (let ((name (car ld))
        (nval (sum-numbers (cdr ld)))
        (sval (extract-names-simple (cdr ld))))
    (cons name
          (cond ((nlistp sval) (list nval))
                ((zerop nval) sval)
                (t (cons nval sval))))))


(disable sum-loading)


(defn sum-loadings (lds)
  (if (nlistp lds)
      nil
    (cons (sum-loading (car lds))
          (sum-loadings (cdr lds)))))


(disable sum-loadings)


;;; FIX-LOADINGS transfers loadings from synonyms to their parent synonym.
;;; It then sums numeric loadings and removes duplicates from name loadings.
```

```
(defn fix-loadings (slist lds)
  (if (nlistp slist)
      (sum-loadings lds)
    (if (or (litatom (cdar slist)) (indexp (cdar slist)))
        (fix-loadings (cdr slist)
                      (add-loading-simple (cdar slist)
                                          (value (caar slist) lds)
                                          (delete-module (caar slist) lds)))
      (fix-loadings (cdr slist) lds))))

(disable fix-loadings)

;;; FIX-DRIVES replaces name drive values with the parent synonym.

(defn fix-drives (drs slist)
  (if (nlistp drs)
      nil
    (if (or (litatom (cdar drs)) (indexp (cdar drs)))
        (cons (cons (caar drs) (parent-synonym-simple (cdar drs) slist))
              (fix-drives (cdr drs) slist))
      (cons (car drs) (fix-drives (cdr drs) slist)))))

(disable fix-drives)

;;; NET-DRIVES subtracts loads from drive values, returning F if a signal is
;;; overloaded.

(defn net-drives (drs lds)
  (cond ((nlistp drs) nil)
        ((numberp (cdar drs))
         (let ((nld (sum-numbers (value (caar drs) lds))))
           (if (lessp (cdar drs) nld)
               F
             (let ((r (net-drives (cdr drs) lds)))
               (if r
                   (cons (cons (caar drs) (difference (cdar drs) nld))
                         r)
                 F)))))
        (t (let ((r (net-drives (cdr drs) lds)))
             (if r
                 (cons (car drs) r)
               F)))))
```

```
(disable net-drives)


;;; EXTERNAL-LOADING removes internal signal names (names not in
;;; external-names) from ld-lst (a list of loadings).


(defn external-loading (ld-lst external-names)
  (cond ((nlistp ld-lst) ld-lst)
        ((and (or (litatom (car ld-lst)) (indexp (car ld-lst)))
              (not (member (car ld-lst) external-names)))
         (external-loading (cdr ld-lst) external-names))
        (t (cons (car ld-lst)
                 (external-loading (cdr ld-lst) external-names)))))


(disable external-loading)


(defn collect-loadings-simple (names loadings external-names)
  ;; Each value in the alist LOADINGS should be a list, possibly empty.
  (if (nlistp names)
      nil
    (if (lookup-module (car names) loadings)
        (let ((r (external-loading (value (car names) loadings)
                                   external-names)))
          (cons
           (cond ((nlistp r) 0)
                 ((nlistp (cdr r)) (car r))
                 (t r))
           (collect-loadings-simple (cdr names) loadings external-names)))
      (cons 0
            (collect-loadings-simple (cdr names) loadings external-names)))))
```

```
(disable collect-loadings-simple)


;;;  COLLECT-DRIVES-SIMPLE collects drive values for module outputs
;;;  NAMES, which are expected to be a subset of EXTERNAL-NAMES
;;;  (module inputs and outputs).  DRIVES should be an alist, and, on
;;;  the initial call, each value in it should be either a number or
;;;  the parent synonym of the corresponding key.
;;;
;;;  The code beginning "(let ((s (parent-synonym-simple v drives)))"
;;;  handles cases like
;;;    (FANOUT-4 (A)
;;;             (Z0 Z1 Z2 Z3)
;;;             ((AA (AA) B-BUF (A))
;;;              (G0 (Z0) ID (AA))
;;;              (G1 (Z1) ID (AA))
;;;              (G2 (Z2) ID (AA))
;;;              (G3 (Z3) ID (AA)))
;;;             NIL)

;;;  where some outputs are synonyms for an internal signal.  The
;;;  first such output (Z0) gets the drive value of the internal
;;;  signal (AA), and the other outputs are marked as synonyms of the
;;;  first output instead of the internal signal.  In the if-test true
;;;  branch (equal s v), internal signal v is the parent synonym, and
;;;  its drive value will be numeric (every parent synonym either has
;;;  a numeric drive value or is a module input).  The first output
;;;  (car names) is given v's drive value and made the parent synonym
;;;  of v.  In the if-test false branch, s and v are different only
;;;  when DRIVES has been modified by the if-test true branch; s is a
;;;  member of the original NAMES list, thus an external name.


(defn collect-drives-simple (names drives external-names)
  (if (nlistp names)
      nil
    (let ((v (value (car names) drives)))
      (if (or (not (or (litatom v) (indexp v)))
              (member v external-names))
          (cons v (collect-drives-simple (cdr names) drives external-names))
        (let ((s (parent-synonym-simple v drives)))
          (if (equal s v)
              (cons (value v drives)
                    (collect-drives-simple
                     (cdr names)
                     (cons (cons v (car names))
                           (cons (cons (car names) (value v drives))
                                 (delete-module (car names)
                                                (delete-module v drives))))
                     external-names))
            (cons s (collect-drives-simple (cdr names)
                                           drives external-names))))))))
```

```
(disable collect-drives-simple)
```

```
(defn loadings-and-drives-simple (flg x0 x1 x2 netlist lds-table)
  (case
    flg
    (0 (let ((fn x0)
             (inputs x1)
             (outputs x2))

         (cond
          ((primp fn)
           (let ((arg-map (append (pairlist (primp2 fn 'inputs) inputs)
                                  (pairlist (primp2 fn 'outputs)
                                            outputs))))
             (cons (fix-dependent-lds (primp2 fn 'loadings) arg-map)
                   (fix-dependent-lds (primp2 fn 'drives) arg-map))))


          ((and (lookup-module 'loadings
                               (cdr (lookup-module fn lds-table)))
                (lookup-module 'drives
                               (cdr (lookup-module fn lds-table))))
           (let ((module (lookup-module fn netlist)))
             (let ((arg-map (append (pairlist (module-inputs module)
                                              inputs)
                                    (pairlist (module-outputs module)
                                              outputs))))
               (cons (fix-dependent-lds
                      (cdr (lookup-module 'loadings
                                          (cdr (lookup-module fn
                                                              lds-table))))
                      arg-map)
                     (fix-dependent-lds
                      (cdr (lookup-module 'drives
                                          (cdr (lookup-module fn
                                                              lds-table))))
                      arg-map)))))

          (t (let ((module (lookup-module fn netlist)))
               (if (listp module)
                   (let ((m-ins  (module-inputs      module))
                         (m-outs (module-outputs     module))
                         (m-occs (module-occurrences module)))

                     (let ((o-lds-drs (loadings-and-drives-simple
                                       1 m-occs nil nil
                                       (delete-module fn netlist)
                                       lds-table)))
                       (if o-lds-drs
                           (let ((o-lds (car o-lds-drs))
                                 (o-drs (cdr o-lds-drs))
                                 (arg-map (append
                                           (pairlist m-ins inputs)
                                           (pairlist m-outs outputs))))
                             (cons (fix-dependent-lds
                                    (collect-loadings-simple m-ins o-lds
```

```
                                                              m-outs)
                                      arg-map)
                                     (fix-dependent-lds
                                      (collect-drives-simple m-outs o-drs
                                                             (append m-ins
                                                                     m-outs))
                                      arg-map)))
                           f)))

                   f))))))

    (1 (let ((body x0)
             (loadings x1)
             (drives x2))

         (if (nlistp body)
             (let ((fixed-drs (fix-drives drives drives)))
               (let ((fixed-lds (fix-loadings fixed-drs loadings)))
                 (let ((net-drs (net-drives fixed-drs fixed-lds)))
                   (if (and fixed-lds net-drs)
                       (cons fixed-lds net-drs)
                     f))))

           (let ((occ (car body)))
             (let ((o-outs (occ-outputs  occ))
                   (o-fn   (occ-function occ))
                   (o-ins  (occ-inputs   occ)))

               (let ((o-lds-drs
                      (loadings-and-drives-simple 0 o-fn o-ins o-outs
                                                  netlist lds-table)))
                 (if o-lds-drs
                     (loadings-and-drives-simple
                      1 (cdr body)
                      (add-loading-simples o-ins (car o-lds-drs) loadings)
                      (append (pairlist o-outs (cdr o-lds-drs))
                              drives)
                      netlist lds-table)
                   f)))))))

    (otherwise f))

  ((ord-lessp (cons (add1 (count netlist)) (count x0)))))


(disable loadings-and-drives-simple)
```

```
(defn netlist-loadings-and-drives-simple (netlist)
  (if (nlistp netlist)
      nil
    (let ((r (netlist-loadings-and-drives-simple (cdr netlist)))
          (module (car netlist)))
      (if r
          (let ((m-name (module-name    module))
                (m-ins  (module-inputs  module))
                (m-outs (module-outputs module)))
            (let ((m-lds-drs (loadings-and-drives-simple 0 m-name m-ins m-outs
                                                         netlist r)))

              (if m-lds-drs
                  (cons (list m-name
                              (cons 'loadings (car m-lds-drs))
                              (cons 'drives (cdr m-lds-drs)))
                        r)
                f)))
        f))))

(disable netlist-loadings-and-drives-simple)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Top Level Predicates
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  Our circuit predicates should be used in a specific order.  The
;;;  predicate NETLIST-SYNTAX-SIMPLE-OKP must be used to ensure that
;;;  the basic syntax and arities of all modules in a netlist are
;;;  correct; our other predicates depend on this.
```

```
(defn top-level-predicate-simple (netlist)
  (if (not (netlist-syntax-simple-okp netlist))
      'netlist-syntax-simple-okp-error
  (if (not (simple-dependency-table netlist))
      'dependency-table-simple-error
  (if (not (netlist-type-check-simple-okp netlist))
      'netlist-type-check-simple-okp-error
  (if (not (netlist-state-types-simple netlist))
      'netlist-state-types-simple-error
  (if (not (netlist-loadings-and-drives-simple netlist))
      'netlist-loadings-and-drives-simple-error
      t))))))


#|


(setq d2 '((d2 (in0)
               (out0)
               ((g (out0) b-and (in0 out0)))
               nil)))


(setq d3 (list (list 'd3
                     '(in0)
                     (list (index 'out0 3))
                     (list (list 'g
                                 (list (index 'out0 3))
                                 'b-and
                                 (list 'in0
                                       (index 'out0 3))))
                     nil)))


(defn d4 ()
  (list (list 'd4
              '(in0)
              (list (index 'out0 3))
              (list (list 'g
                          (list (index 'out0 3))
                          'b-and
                          (list 'in0
                                (index 'out0 3))))
              nil)))


(netlist-syntax-simple-okp d2 10 nil)

(out-depends-simple 0 'd2 '((a)) d2 nil)
```

```
(setq ex-net
      '((m1 (clk)
            (out1 out2)
            ((o1 (out1 out2) fd1 (out1 clk)))
            o1)))


(top-level-predicate ex-net)

(netlist-loadings-and-drives-simple ex-net)
  = '((M1 (LOADINGS 1) (DRIVES 9 10)))



(setq ex-net '((m3 (in0 clk) (out1 out2)
                   ((o1 (oi1) b-not (in0))
                    (o2 (out1) b-not (in0))
                    (o5 (out2 waste) fd1 (oi2 clk))
                           ; oi2 is ok here because out2 does not
                           ; depend on oi2.  Out2 depends only on
                           ; fd1's internal state before update.
                    (o4 (ii1) b-and (out2 oi1))
                    (o3 (oi2) b-and (in0 ii1)))
                   o5)))


(netlist-syntax-simple-okp ex-net 10 nil)

(out-depends-simple 0 'm3 '((new-in) (new-clk)) ex-net nil)


(arg-types-simple-okp 'm3 '(boolp clk) ex-net)

(arg-types-simple-okp 'm3 '(clk boolp) ex-net)

(arg-types-simple-okp 'm3 '(boolp boolp) ex-net)

(netlist-type-table-simple ex-net)

(netlist-type-check-simple-okp ex-net)


(netlist-state-types-simple ex-net)

; in the following, first 2 are ok and the remaining 3 are not

(list (apply-state-simple-okp 'm3 f ex-net)
      (apply-state-simple-okp 'm3 t ex-net)
      (apply-state-simple-okp 'm3 (x) ex-net)
      (apply-state-simple-okp 'm3 (list t) ex-net)
      (apply-state-simple-okp 'm3 nil ex-net))
```

```
(netlist-loadings-and-drives-simple ex-net)
  = '((M3 (LOADINGS 5 1) (DRIVES 10 9)))
```

```
(setq ex-net '((m3 (in0 clk) (out1 out2)
                  ((o1 (is1) id (in0))
                   (o2 (is2) id (clk))
                   (o3 (is3 is4) fd1 (is1 is2))
                   ; oi2 is ok here because out2 does not depend on
                   ; oi2.  Out2 depends only on fd1's internal state
                   ; before update.
                   (o4 (out1) id (is3))
                   (o5 (out2) id (is4)))
                  o3)))
```

```
(arg-types-simple-okp 'm3 '(boolp clk) ex-net)

(arg-types-simple-okp 'm3 '(clk boolp) ex-net)

(arg-types-simple-okp 'm3 '(boolp boolp) ex-net)

(netlist-type-table-simple ex-net)

(netlist-type-check-simple-okp ex-net)
```

```
(netlist-loadings-and-drives-simple ex-net)
  = '((M3 (LOADINGS 1 1) (DRIVES 10 10)))
```

```
(setq ex-net '((m1 (in0) (out0)
                  ((o1 (is1) id (in0))
                   (o2 (is2) id (is1))
                   (o3 (is3) id (is2))
                   (o4 (out0) id (is3)))
                  nil)))
```

```
(arg-types-simple-okp 'm1 '(boolp) ex-net)

(arg-types-simple-okp 'm1 '(clk) ex-net)

(arg-types-simple-okp 'm1 '(parametric) ex-net)

(netlist-type-table-simple ex-net)

(netlist-type-check-simple-okp ex-net)
```

```
(netlist-state-types-simple ex-net)

; in the following, the first is ok and the other 2 are not

(list (apply-state-simple-okp 'm1 nil ex-net)
      (apply-state-simple-okp 'm1 t ex-net)
      (apply-state-simple-okp 'm1 (x) ex-net))


(netlist-loadings-and-drives-simple ex-net)
 '((M1 (LOADINGS OUT0) (DRIVES IN0)))



(setq test-net '((test () (a b)
                       ((inv1 (a) b-not   (b))
                        (inv2 (b) b-not   (a)))
                       nil)))


(netlist-syntax-simple-okp test-net 10 nil)

(out-depends-simple 0 'test '() test-net nil)

(netlist-state-input-simple-okp test-net (dependency-table test-net))




(setq netlist '((m2 (clk en0 en1 sel0 sel1 d0 d1) (q)
                    ((occ0 (q0) m1     (clk en0 sel0 d0 q))
                     (occ1 (q1) m1     (clk en1 sel1 d1 q))
                     (wire (q)  t-wire (q0 q1)))
                    (occ0 occ1))
                (m1 (clk en sel d q) (q)
                    ((mux   (b)    b-if  (sel d q))
                     (latch (a an) fd1   (b clk))
                     (tbuf  (q)    t-buf (en a)))
                    latch)))



(setq args '((a) (b) (c) (d) (e)))


(out-depends-simple 0 'm1 args netlist module-database)



(setq args-if2 '((c) (v0 v1) (x0) (v0 w0) (y0 y1)))

(out-depends-simple 0 'b-if2 args-if2 nil module-database)
```

```
(setq args-core6
      '((c) (a0) (a1) (b0) (b1) (zero)
        (mpg-0) (mpg-1) (mpg-2) (mpg-3) (mpg-4) (mpg-5) (mpg-6)
        (op-0) (op-1) (op-2) (op-3)))


(output-dependencies-simple (index 'core-alu 6) args-core6 test)




(setq args-core108
      '((c) (a0) (a1) (a2) (a3) (b0) (b1) (b2) (b3) (zero)
        (mpg-0) (mpg-1) (mpg-2) (mpg-3) (mpg-4) (mpg-5) (mpg-6)
        (op-0) (op-1) (op-2) (op-3)))


(setq arg-types-core108
      '(boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp))


(setq netlist-4 (core-alu$netlist (make-tree 4)))

(setq module-database (dependency-table netlist-4))


(output-dependencies-simple (index 'core-alu 108) args-core108 netlist-4)

(netlist-syntax-simple-okp netlist-4 20 nil)


(netlist-type-table-simple netlist-4)

(arg-types-simple-okp (index 'core-alu 108) arg-types-core108 netlist-4)

(netlist-type-check-simple-okp netlist-4)
```

```
(netlist-loadings-and-drives-simple netlist-4)
= (LIST (LIST (INDEX 'CORE-ALU 108)
              '(LOADINGS 5 6 4 4 7 3 3 3 5 5 4 4 4 4 4 4 4 8 7 6 7)
              '(DRIVES 10 10 10 9 9 9 9))
        (LIST (INDEX 'TV-ALU-HELP 108)
              '(LOADINGS 3 4 4 4 4 3 3 3 3 4 4 4 4 4 4)
              '(DRIVES 10 10 10 10 10 10))
        (LIST (INDEX 'TV-ALU-HELP 6)
              '(LOADINGS 2 4 4 3 3 2 2 2 2 2 2 2)
              '(DRIVES 10 10 10 10))
        (LIST (INDEX 'TV-ALU-HELP 1)
              '(LOADINGS 1 4 3 1 1 1 1 1 1 1)
              '(DRIVES 7 8 10))
        '(ALU-CELL (LOADINGS 1 4 3 1 1 1 1 1 1 1)
                   (DRIVES 7 8 10))
        '(P-CELL (LOADINGS 1 1 1 1 1 1)
                 (DRIVES 10))
        '(G-CELL (LOADINGS 1 1 1 1 1 1)
                 (DRIVES 10))
        (LIST (INDEX 'V-BUF 7)
              '(LOADINGS 1 1 1 1 1 1 1)
              '(DRIVES 10 10 10 10 10 10 10))
        '(T-CARRY (LOADINGS 1 1 1) (DRIVES 10))
        '(CARRY-OUT-HELP (LOADINGS 2 2 2 2 2 2 2)
                         (DRIVES 10))
        '(OVERFLOW-HELP (LOADINGS 1 2 2 2 2 2 2 2)
                        (DRIVES 10))
        (LIST (INDEX 'TV-SHIFT-OR-BUF 108)
              '(LOADINGS 1 1 2 2 2 1 1 4 3 2 3)
              '(DRIVES 10 10 10 10))
        (LIST (INDEX 'TV-IF 108)
              '(LOADINGS 1 1 1 1 1 1 1 1 1)
              '(DRIVES 10 10 10 10))
        (LIST (INDEX 'TV-IF 6)
              '(LOADINGS 4 1 1 1 1)
              '(DRIVES 10 10))
        (LIST (INDEX 'TV-IF 1)
              '(LOADINGS 2 1 1)
              '(DRIVES 10))
        (LIST (INDEX 'TV-ZEROP 108)
              '(LOADINGS 1 1 1 1)
              '(DRIVES 10))
        (LIST (INDEX 'T-OR 108)
              '(LOADINGS 1 1 1 1)
              '(DRIVES 10))
        (LIST (INDEX 'T-NOR 6)
              '(LOADINGS 1 1)
              '(DRIVES 10))
        '(B-BUF (LOADINGS 1) (DRIVES 10)))
```

```
(setq args-16
      '((C-in) (a0) (a1) (a2) (a3) (a4) (a5) (a6) (a7) (a8) (a9) (a10)
        (a11) (a12) (a13) (a14) (a15) (b0) (b1) (b2) (b3) (b4) (b5) (b6) (b7)
        (b8) (b9) (b10) (b11) (b12) (b13) (b14) (b15) zero (mpg0) (mpg1)
        (mpg2) (mpg3) (mpg4) (mpg5) (mpg6) (op0) (op1) (op2) (op3)))


(setq arg-types-16
      '(boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp boolp boolp boolp boolp boolp boolp))


(setq netlist-16 (core-alu$netlist (make-tree 16)))


(netlist-syntax-simple-okp netlist-16 30 nil)


(output-dependencies-simple (index 'core-alu 1826150832) args-16 netlist-16)


(netlist-type-table-simple netlist-16)

(arg-types-simple-okp (index 'core-alu 1826150832) arg-types-16 netlist-16)

(netlist-type-check-simple-okp netlist-16)
```

```
(netlist-loadings-and-drives-simple netlist-16)
= (LIST (LIST (INDEX 'CORE-ALU 1826150832)
              '(LOADINGS 7 6 4 4 4 4 4 4 4 4 4 4 4 4 4 4 7 3 3 3 3 3
                         3 3 3 3 3 3 3 3 3 3 3 5 5 2 2 2 2 2 2 2 2 8 7 6
                         7)
              '(DRIVES 10 10 10 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9))
        (LIST (INDEX 'TV-ALU-HELP 1826150832)
              '(LOADINGS 5 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 3 3 3 3 3
                         3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2)
              '(DRIVES 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
                       10 10 10))
        (LIST (INDEX 'TV-ALU-HELP 27864)
              '(LOADINGS 4 4 4 4 4 4 4 4 4 4 3 3 3 3 3 3 3 3 1 1 1 1 1
                         1 1)
              '(DRIVES 10 10 10 10 10 10 10 10 10 10))
        (LIST (INDEX 'TV-ALU-HELP 108)
              '(LOADINGS 3 4 4 4 4 3 3 3 3 4 4 4 4 4 4 4)
              '(DRIVES 10 10 10 10 10 10))
        (LIST (INDEX 'TV-ALU-HELP 6)
              '(LOADINGS 2 4 4 3 3 2 2 2 2 2 2 2)
              '(DRIVES 10 10 10 10))
        (LIST (INDEX 'TV-ALU-HELP 1)
              '(LOADINGS 1 4 3 1 1 1 1 1 1 1)
              '(DRIVES 7 8 10))
        '(ALU-CELL (LOADINGS 1 4 3 1 1 1 1 1 1 1)
                   (DRIVES 7 8 10))
        '(P-CELL (LOADINGS 1 1 1 1 1 1)
                 (DRIVES 10))
        '(G-CELL (LOADINGS 1 1 1 1 1 1)
                 (DRIVES 10))
        (LIST (INDEX 'V-BUF 7)
              '(LOADINGS 1 1 1 1 1 1 1)
              '(DRIVES 10 10 10 10 10 10 10))
        '(T-CARRY (LOADINGS 1 1 1) (DRIVES 10))
        '(CARRY-OUT-HELP (LOADINGS 2 2 2 2 2 2 2)
                         (DRIVES 10))
        '(OVERFLOW-HELP (LOADINGS 1 2 2 2 2 2 2 2)
                        (DRIVES 10))
        (LIST (INDEX 'TV-SHIFT-OR-BUF 1826150832)
              '(LOADINGS 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 4 3 2
                         3)
              '(DRIVES 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
                       10))
        (LIST (INDEX 'TV-IF 1826150832)
              '(LOADINGS 4 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                         1 1 1 1 1 1 1 1 1 1 1)
              '(DRIVES 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
                       10))
        (LIST (INDEX 'TV-IF 27864)
              '(LOADINGS 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
              '(DRIVES 10 10 10 10 10 10 10 10))
        (LIST (INDEX 'TV-IF 108)
              '(LOADINGS 1 1 1 1 1 1 1 1 1)
              '(DRIVES 10 10 10 10))
```

```
       (LIST (INDEX 'TV-IF 6)
             '(LOADINGS 4 1 1 1 1)
             '(DRIVES 10 10))
       (LIST (INDEX 'TV-IF 1)
             '(LOADINGS 2 1 1)
             '(DRIVES 10))
       (LIST (INDEX 'TV-ZEROP 1826150832)
             '(LOADINGS 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
             '(DRIVES 10))
       (LIST (INDEX 'T-OR 1826150832)
             '(LOADINGS 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
             '(DRIVES 10))
       (LIST (INDEX 'T-NOR 27864)
             '(LOADINGS 1 1 1 1 1 1 1 1)
             '(DRIVES 10))
       (LIST (INDEX 'T-OR 108)
             '(LOADINGS 1 1 1 1)
             '(DRIVES 10))
       (LIST (INDEX 'T-NOR 6)
             '(LOADINGS 1 1)
             '(DRIVES 10))
       '(B-BUF (LOADINGS 1) (DRIVES 10)))
```

```
(setq netlist-32 (core-alu$netlist (make-tree 32)))
```

```
(top-level-predicate netlist-32)
```

```
(output-dependencies-simple (caar netlist-32)
                            (listify (module-inputs (car netlist-32)))
                            netlist-32)
```

```
(netlist-loadings-and-drives-simple netlist-32)
= (LIST (LIST (INDEX 'CORE-ALU 7843258104655491936)
              '(LOADINGS 8 6 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
                         4 4 4 4 4 4 4 4 4 4 7 3 3 3 3 3 3 3 3 3 3 3
                         3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 5 5
                         4 4 4 4 4 4 4 8 7 6 7)
              '(DRIVES 10 10 10 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
                       9 9 9 9 9 9 9 9 9 9 9 9 9 9))
        (LIST (INDEX 'TV-ALU-HELP
                     7843258104655491936)
              '(LOADINGS 6 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
                         4 4 4 4 4 4 4 4 4 4 4 3 3 3 3 3 3 3 3 3 3 3
                         3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4
                         4 4 4 4 4 4)
              '(DRIVES 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
                       10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
                       10 10 10 10))
        (LIST (INDEX 'TV-ALU-HELP 1826150832)
              '(LOADINGS 5 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 3 3 3 3 3
                         3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2)
              '(DRIVES 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
                       10 10 10))
        (LIST (INDEX 'TV-ALU-HELP 27864)
              '(LOADINGS 4 4 4 4 4 4 4 4 4 3 3 3 3 3 3 3 3 1 1 1 1 1
                         1 1)
              '(DRIVES 10 10 10 10 10 10 10 10 10 10))
        (LIST (INDEX 'TV-ALU-HELP 108)
              '(LOADINGS 3 4 4 4 4 3 3 3 3 4 4 4 4 4 4 4)
              '(DRIVES 10 10 10 10 10 10))
        (LIST (INDEX 'TV-ALU-HELP 6)
              '(LOADINGS 2 4 4 3 3 2 2 2 2 2 2 2)
              '(DRIVES 10 10 10 10))
        (LIST (INDEX 'TV-ALU-HELP 1)
              '(LOADINGS 1 4 3 1 1 1 1 1 1 1)
              '(DRIVES 7 8 10))
        '(ALU-CELL (LOADINGS 1 4 3 1 1 1 1 1 1 1)
                   (DRIVES 7 8 10))
        '(P-CELL (LOADINGS 1 1 1 1 1 1)
                 (DRIVES 10))
        '(G-CELL (LOADINGS 1 1 1 1 1 1)
                 (DRIVES 10))
        (LIST (INDEX 'V-BUF 7)
              '(LOADINGS 1 1 1 1 1 1 1)
              '(DRIVES 10 10 10 10 10 10 10))
        '(T-CARRY (LOADINGS 1 1 1) (DRIVES 10))
        '(CARRY-OUT-HELP (LOADINGS 2 2 2 2 2 2 2 2)
                         (DRIVES 10))
        '(OVERFLOW-HELP (LOADINGS 1 2 2 2 2 2 2 2)
                        (DRIVES 10))
        (LIST (INDEX 'TV-SHIFT-OR-BUF
                     7843258104655491936)
              '(LOADINGS 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
                         2 2 2 2 2 2 2 2 2 2 2 1 1 4 3 2 3)
              '(DRIVES 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

```
                  10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
                  10 10))
      (LIST (INDEX 'TV-IF 7843258104655491936)
            '(LOADINGS 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                       1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                       1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
            '(DRIVES 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
                     10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
                     10 10))
      (LIST (INDEX 'TV-IF 1826150832)
            '(LOADINGS 4 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                       1 1 1 1 1 1 1 1 1 1 1)
            '(DRIVES 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
                     10))
      (LIST (INDEX 'TV-IF 27864)
            '(LOADINGS 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
            '(DRIVES 10 10 10 10 10 10 10 10))
      (LIST (INDEX 'TV-IF 108)
            '(LOADINGS 1 1 1 1 1 1 1 1 1)
            '(DRIVES 10 10 10 10))
      (LIST (INDEX 'TV-IF 6)
            '(LOADINGS 4 1 1 1 1)
            '(DRIVES 10 10))
      (LIST (INDEX 'TV-IF 1)
            '(LOADINGS 2 1 1)
            '(DRIVES 10))
      (LIST (INDEX 'TV-ZEROP 7843258104655491936)
            '(LOADINGS 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                       1 1 1 1 1 1 1 1 1 1)
            '(DRIVES 10))
      (LIST (INDEX 'T-NOR 7843258104655491936)
            '(LOADINGS 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                       1 1 1 1 1 1 1 1 1 1)
            '(DRIVES 10))
      (LIST (INDEX 'T-OR 1826150832)
            '(LOADINGS 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
            '(DRIVES 10))
      (LIST (INDEX 'T-NOR 27864)
            '(LOADINGS 1 1 1 1 1 1 1 1)
            '(DRIVES 10))
      (LIST (INDEX 'T-OR 108)
            '(LOADINGS 1 1 1 1)
            '(DRIVES 10))
      (LIST (INDEX 'T-NOR 6)
            '(LOADINGS 1 1)
            '(DRIVES 10))
      '(B-BUF (LOADINGS 1) (DRIVES 10)))
```

```
(defn dummy (x) 0)
```

```
(defn netlist-syntax-simple-okp2 (netlist token-size)
  (if (nlistp netlist)

      (equal netlist nil)

      (if (module-syntax-simple-okp netlist token-size nil)
          (netlist-syntax-simple-okp2 (cdr netlist) token-size)
          (and (dummy (caar netlist)) f))))



|#


#|


(setq ex-net '((m1 (in1 in2 clk) (out1 out2 out3 out4)
                   ((o1 (out1 out2 out3 out4)
                        fd11
                        (in1 in2 clk)))
                   o1)
               (fd11 (in1 in2 clk)
                     (out1 out2 out3 out4)
                     ((o1 (out1 out2) fd1 (in1 clk))
                      (o2 (out3 out4) fd1 (in2 clk)))
                     (o1 o2))))


(netlist-state-types-simple ex-net)

; in the following, the first 4 are ok and the remaining 6 are not

(list (apply-state-simple-okp 'm1 (list f f) ex-net)
      (apply-state-simple-okp 'm1 (list f t) ex-net)
      (apply-state-simple-okp 'm1 (list t f) ex-net)
      (apply-state-simple-okp 'm1 (list t t) ex-net)
      (apply-state-simple-okp 'm1 (list t (x)) ex-net)
      (apply-state-simple-okp 'm1 (list (x) f) ex-net)
      (apply-state-simple-okp 'm1 (list t f t) ex-net)
      (apply-state-simple-okp 'm1 (list f) ex-net)
      (apply-state-simple-okp 'm1 nil ex-net)
      (apply-state-simple-okp 'm1 (x) ex-net))


(netlist-loadings-and-drives-simple ex-net)
 '((M1 (LOADINGS 1 1 2)
       (DRIVES 10 10 10 10))
   (FD11 (LOADINGS 1 1 2)
         (DRIVES 10 10 10 10)))
```

```
(setq ex-net
      '((m1 (in1 in2 in3 in4 clk) (out1 out2 out3 out4)
            ((o1 (out1 w1) fd1 (in1 clk))
             (o2 (out2 w2) fd1 (in2 clk))
             (o3 (out3 w3) fd1 (in3 clk))
             (o4 (out4 w4) fd1 (in4 clk)))
            (o1 o2 o3 o4))))


(netlist-state-types-simple ex-net)

; in the following the first 4 are ok and the remaining 6 are not

(list (apply-state-simple-okp 'm1 (list f f f f) ex-net)
      (apply-state-simple-okp 'm1 (list f t f t) ex-net)
      (apply-state-simple-okp 'm1 (list t f t f) ex-net)
      (apply-state-simple-okp 'm1 (list t t t t) ex-net)
      (apply-state-simple-okp 'm1 (list t f (x) t) ex-net)
      (apply-state-simple-okp 'm1 (list (x) t t t) ex-net)
      (apply-state-simple-okp 'm1 (list t f t f t) ex-net)
      (apply-state-simple-okp 'm1 (list f t f) ex-net)
      (apply-state-simple-okp 'm1 nil ex-net)
      (apply-state-simple-okp 'm1 (x) ex-net))


(netlist-loadings-and-drives-simple ex-net)
 '((M1 (LOADINGS 1 1 1 1 4)
       (DRIVES 10 10 10 10)))



(setq ex-net
      '((m1 (in1 in2 in3 in4 clk) (out1 out2)
            ((o1 (is1 w1) fd1 (in1 clk))
             (o2 (is2 w2) fd1 (in2 clk))
             (o3 (out1) b-and (is1 is2))
             (o4 (is3 w3) fd1 (in3 clk))
             (o5 (is4 w4) fd1 (in4 clk))
             (o6 (out2) b-and (is3 is4)))
            (o1 o2 o4 o5))))


(netlist-state-types-simple ex-net)

; in the following the first 4 are ok and the remaining 6 are not
```

```
(list (apply-state-simple-okp 'm1 (list f f f f) ex-net)
      (apply-state-simple-okp 'm1 (list f t f t) ex-net)
      (apply-state-simple-okp 'm1 (list t f t f) ex-net)
      (apply-state-simple-okp 'm1 (list t t t t) ex-net)
      (apply-state-simple-okp 'm1 (list t f (x) t) ex-net)
      (apply-state-simple-okp 'm1 (list (x) t t t) ex-net)
      (apply-state-simple-okp 'm1 (list t f t f t) ex-net)
      (apply-state-simple-okp 'm1 (list f t f) ex-net)
      (apply-state-simple-okp 'm1 nil ex-net)
      (apply-state-simple-okp 'm1 (x) ex-net))


(netlist-loadings-and-drives-simple ex-net)
 '((M1 (LOADINGS 1 1 1 1 4)
       (DRIVES 10 10)))



(setq ex-net
      '((m1
         (read-a0 read-a1 read-a2 read-a3
          write-b0 write-b1 write-b2 write-b3
          wen
          d0  d1  d2  d3  d4  d5  d6  d7
          d8  d9  d10 d11 d12 d13 d14 d15
          d16 d17 d18 d19 d20 d21 d22 d23
          d24 d25 d26 d27 d28 d29 d30 d31)
         (o0  o1  o2  o3  o4  o5  o6  o7
          o8  o9  o10 o11 o12 o13 o14 o15
          o16 o17 o18 o19 o20 o21 o22 o23
          o24 o25 o26 o27 o28 o29 o30 o31)
         ((o1
           (o0  o1  o2  o3  o4  o5  o6  o7
            o8  o9  o10 o11 o12 o13 o14 o15
            o16 o17 o18 o19 o20 o21 o22 o23
            o24 o25 o26 o27 o28 o29 o30 o31)
           dp-ram-16x32
           (read-a0 read-a1 read-a2 read-a3
            write-b0 write-b1 write-b2 write-b3
            wen
            d0  d1  d2  d3  d4  d5  d6  d7
            d8  d9  d10 d11 d12 d13 d14 d15
            d16 d17 d18 d19 d20 d21 d22 d23
            d24 d25 d26 d27 d28 d29 d30 d31)))
         o1)))


(netlist-state-types-simple ex-net)

; in the following, the first 2 are ok and the remaining 5 are not
```

```
(list (apply-state-simple-okp 'm1 (make-ram-state (make-tree 16) 32 t) ex-net)
      (apply-state-simple-okp 'm1 (make-ram-state (make-tree 16) 32 f) ex-net)
      (apply-state-simple-okp 'm1 (make-ram-state (make-tree 8) 32 f) ex-net)
      (apply-state-simple-okp 'm1 (make-ram-state (make-tree 16) 8 f) ex-net)
      (apply-state-simple-okp
       'm1 (list (make-ram-state (make-tree 16) 32 t)) ex-net)
      (apply-state-simple-okp 'm1 (x) ex-net)
      (apply-state-simple-okp 'm1 nil ex-net))


(netlist-loadings-and-drives-simple ex-net)
 '((M1 (LOADINGS 2 2 2 2 2 2 2 2 2 4 1 1 1 1 1 1 1 1 1 1 1 1
                 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
       (DRIVES 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
               2 2 2 2 2 2)))



(setq ex-net
      '((m1
         (clk
          read-a0 read-a1 read-a2 read-a3
          write-b0 write-b1 write-b2 write-b3
          wen
          d0  d1  d2  d3  d4  d5  d6  d7
          d8  d9  d10 d11 d12 d13 d14 d15
          d16 d17 d18 d19 d20 d21 d22 d23
          d24 d25 d26 d27 d28 d29 d30 d31)
         (o0  o1  o2  o3  o4  o5  o6  o7
          o8  o9  o10 o11 o12 o13 o14 o15
          o16 o17 o18 o19 o20 o21 o22 o23
          o24 o25 o26 o27 o28 o29 o30 o31)
         ((o0 (is w2) fd1 (d0 clk))
          (o1
           (o0  o1  o2  o3  o4  o5  o6  o7
            o8  o9  o10 o11 o12 o13 o14 o15
            o16 o17 o18 o19 o20 o21 o22 o23
            o24 o25 o26 o27 o28 o29 o30 o31)
           dp-ram-16x32
           (read-a0 read-a1 read-a2 read-a3
            write-b0 write-b1 write-b2 write-b3
            wen
            is  d1  d2  d3  d4  d5  d6  d7
            d8  d9  d10 d11 d12 d13 d14 d15
            d16 d17 d18 d19 d20 d21 d22 d23
            d24 d25 d26 d27 d28 d29 d30 d31)))
         (o0 o1))))


(netlist-state-types-simple ex-net)

; in the following, the first 4 are ok and remaining 7 are not
```

```
(list (apply-state-simple-okp
       'm1 (list f (make-ram-state (make-tree 16) 32 f)) ex-net)
      (apply-state-simple-okp
       'm1 (list f (make-ram-state (make-tree 16) 32 t)) ex-net)
      (apply-state-simple-okp
       'm1 (list t (make-ram-state (make-tree 16) 32 f)) ex-net)
      (apply-state-simple-okp
       'm1 (list t (make-ram-state (make-tree 16) 32 t)) ex-net)
      (apply-state-simple-okp
       'm1 (list t (make-ram-state (make-tree 15) 32 t)) ex-net)
      (apply-state-simple-okp
       'm1 (list t (make-ram-state (make-tree 16) 33 t)) ex-net)
      (apply-state-simple-okp
       'm1 (list f (make-ram-state (make-tree 16) 32 t) t) ex-net)
      (apply-state-simple-okp
       'm1 (list (make-ram-state (make-tree 16) 32 t)) ex-net)
      (apply-state-simple-okp
       'm1 (list t) ex-net)
      (apply-state-simple-okp
       'm1 (x) ex-net)
      (apply-state-simple-okp
       'm1 nil ex-net))


(netlist-loadings-and-drives-simple ex-net)
 '((M1 (LOADINGS 1 2 2 2 2 2 2 2 2 2 4 1 1 1 1 1 1 1 1
                 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
       (DRIVES 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
               2 2 2 2 2 2)))
```

```
(setq ex-net
      '((m1 (read-a0 read-a1 read-a2 read-a3
             write-b0 write-b1 write-b2 write-b3
             in0
             d0  d1  d2  d3  d4  d5  d6  d7
             d8  d9  d10 d11 d12 d13 d14 d15
             d16 d17 d18 d19 d20 d21 d22 d23
             d24 d25 d26 d27 d28 d29 d30 d31
             clk)
            (out0)
            ((o1 (is1) m5 (in0))
             (o2 (is2) m3 (is1 clk))
             (o3 (is3) m4 (in0 is2))
             (o4 (wen) m5 (is3))
             (o5
              (o0  o1  o2  o3  o4  o5  o6  o7
               o8  o9  o10 o11 o12 o13 o14 o15
               o16 o17 o18 o19 o20 o21 o22 o23
               o24 o25 o26 o27 o28 o29 o30 o31 out0)
              m2
              (read-a0 read-a1 read-a2 read-a3
               write-b0 write-b1 write-b2 write-b3
               wen
               d0  d1  d2  d3  d4  d5  d6  d7
               d8  d9  d10 d11 d12 d13 d14 d15
               d16 d17 d18 d19 d20 d21 d22 d23
               d24 d25 d26 d27 d28 d29 d30 d31
               clk)))
            (o2 o3 o5))
        (m2 (read-a0 read-a1 read-a2 read-a3
             write-b0 write-b1 write-b2 write-b3
             wen
             d0  d1  d2  d3  d4  d5  d6  d7
             d8  d9  d10 d11 d12 d13 d14 d15
             d16 d17 d18 d19 d20 d21 d22 d23
             d24 d25 d26 d27 d28 d29 d30 d31
             clk)
            (o0  o1  o2  o3  o4  o5  o6  o7
             o8  o9  o10 o11 o12 o13 o14 o15
             o16 o17 o18 o19 o20 o21 o22 o23
             o24 o25 o26 o27 o28 o29 o30 o31 out0)
            ((o1
              (o0  o1  o2  o3  o4  o5  o6  o7
               o8  o9  o10 o11 o12 o13 o14 o15
               o16 o17 o18 o19 o20 o21 o22 o23
               o24 o25 o26 o27 o28 o29 o30 o31)
              m6
              (read-a0 read-a1 read-a2 read-a3
               write-b0 write-b1 write-b2 write-b3
               wen
               d0  d1  d2  d3  d4  d5  d6  d7
               d8  d9  d10 d11 d12 d13 d14 d15
               d16 d17 d18 d19 d20 d21 d22 d23
               d24 d25 d26 d27 d28 d29 d30 d31))
```

```
          (o2 (out0) m3 (o31 clk)))
         (o1 o2))
     (m3 (in0 clk) (out0)
         ((o1 (out0) m7 (clk in0)))
         o1)
     (m4 (in1 in2) (out0)
         ((o1 (is1) m5 (in1))
          (o2 (is2 is3 is4 is5) fd11 (is1 in2 clk))
          (o3 (out0) b-and4 (is2 is3 is4 is5)))
         o2)
     (m5 (in0) (out0)
         ((o1 (out0) b-not (in0)))
         nil)
     (m6 (read-a0 read-a1 read-a2 read-a3
          write-b0 write-b1 write-b2 write-b3
          wen
          d0  d1  d2  d3  d4  d5  d6  d7
          d8  d9  d10 d11 d12 d13 d14 d15
          d16 d17 d18 d19 d20 d21 d22 d23
          d24 d25 d26 d27 d28 d29 d30 d31)
         (o0  o1  o2  o3  o4  o5  o6  o7
          o8  o9  o10 o11 o12 o13 o14 o15
          o16 o17 o18 o19 o20 o21 o22 o23
          o24 o25 o26 o27 o28 o29 o30 o31)
         ((o-1
           (o0  o1  o2  o3  o4  o5  o6  o7
            o8  o9  o10 o11 o12 o13 o14 o15
            o16 o17 o18 o19 o20 o21 o22 o23
            o24 o25 o26 o27 o28 o29 o30 o31)
           dp-ram-16x32
           (read-a0 read-a1 read-a2 read-a3
            write-b0 write-b1 write-b2 write-b3
            wen
            d0  d1  d2  d3  d4  d5  d6  d7
            d8  d9  d10 d11 d12 d13 d14 d15
            d16 d17 d18 d19 d20 d21 d22 d23
            d24 d25 d26 d27 d28 d29 d30 d31)))
         o-1)
     (m7 (clk in0) (out0)
         ((o1 (out0 w) fd1 (in0 clk)))
         o1)
     (fd11 (in1 in2 clk)
           (out1 out2 out3 out4)
           ((o1 (out1 out2) fd1 (in1 clk))
            (o2 (out3 out4) fd1 (in2 clk)))
           (o1 o2))))


(setq state
      (list t (list t f) (list (make-ram-state (make-tree 16) 32 t) f)))


(netlist-state-types-simple ex-net) ; repeat with flaws in net
```

```
(apply-state-simple-okp 'm1 state ex-net) ; repeat with flaws in state


(netlist-loadings-and-drives-simple ex-net)
 '((M1 (LOADINGS 2 2 2 2 2 2 2 2 4 1 1 1 1 1 1 1 1
                 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2)
       (DRIVES 10))
   (M2 (LOADINGS 2 2 2 2 2 2 2 2 4 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
       (DRIVES 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
               2 2 2 2 2 1 10))
   (M3 (LOADINGS 1 1) (DRIVES 10))
   (M4 (LOADINGS 2 1) (DRIVES 10))
   (M5 (LOADINGS 2) (DRIVES 10))
   (M6 (LOADINGS 2 2 2 2 2 2 2 2 4 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
       (DRIVES 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
               2 2 2 2 2 2))
   (M7 (LOADINGS 1 1) (DRIVES 10))
   (FD11 (LOADINGS 1 1 2)
         (DRIVES 10 10 10 10)))



(defn list-set-equal (x y)
  (if (or (nlistp x) (nlistp y))
      (and (nlistp x) (nlistp y))
      (if (set-equal (car x) (car y))
          (list-set-equal (cdr x) (cdr y))
          f)))

|#
#|

(setq ex-net1
      '((m1 (in0) (out1 out2)
            ((o1 (out1) b-not (in0))
             (o2 (out2) id (out1)))
            nil)))


(netlist-loadings-and-drives-simple ex-net1)
  = '((M1 (LOADINGS 2) (DRIVES 10 OUT1)))



(setq ex-net2
      '((m1 (in1 in2) (out1 out2)
            ((o1 (is1) id (in1))
             (o2 (out1) b-and (is1 in2))
             (o3 (out2) b-or (is1 in2)))
            nil)))
```

```
(netlist-loadings-and-drives-simple ex-net2)
  = '((M1 (LOADINGS 2 2) (DRIVES 10 10)))



(setq ex-net3
      '((m1 (in0) (out1 out2)
            ((o1 (io1) id (in0))
             (o2 (out1) id (io1))
             (o3 (out2) id (io1)))
            nil)))


(netlist-loadings-and-drives-simple ex-net3)
  = '((M1 (LOADINGS (OUT2 OUT1))
       (DRIVES IN0 IN0)))



(setq ex-net4
      '((m1 (in0) (out1 out2)
            ((o1 (out1) id (in0))
             (o2 (out2) id (out1)))
            nil)))


(netlist-loadings-and-drives-simple ex-net4)
  =  '((M1 (LOADINGS (OUT2 OUT1))
       (DRIVES IN0 IN0)))



(setq ex-net5
      '((m1 (in0) (out1 out2)
            ((o1 (out1) id (in0))
             (o2 (out2) id (in0)))
            nil)))


(netlist-loadings-and-drives-simple ex-net5)
  =  '((M1 (LOADINGS (OUT2 OUT1))
       (DRIVES IN0 IN0)))



(setq ex-net6
      '((m1 () (out0)
            ((o1 (out0) vdd nil))
            nil)))

'((M1 (LOADINGS) (DRIVES 50)))
```

```
(setq ex-net7
      '((m1 () (out0)
            ((o1 (out0) vss nil))
            nil)))

'((M1 (LOADINGS) (DRIVES 50)))
```

```
(setq ex-net8
      '((m1 (a b) (c d)
            ((o1 (c) b-and (a b))
             (o2 (d) id (c)))
            nil)))

 '((M1 (LOADINGS 1 1) (DRIVES 10 C)))
```

```
(setq ex-net9
      '((m1 (in0) (out1 out2 out3 out4)
            ((o1 (out1) id (in0))
             (o2 (out2) b-not (out1))
             (o3 (out3) id (in0))
             (o4 (out4) b-not (in0)))
            nil)))

 '((M1 (LOADINGS (4 OUT3 OUT1))
       (DRIVES IN0 10 IN0 10)))
```

```
(setq ex-net10
      '((m1 (in0) (out0)
            ((o1 (a) id (in0))
             (o2 (b) id (in0))
             (o3 (out0) b-and (a b)))
            nil)))

'((M1 (LOADINGS 2) (DRIVES 10)))
```

```
(setq ex-net11
      '((m1 (in0) (out0)
            ((o1 (a) id (in0))
             (o2 (out0) b-and (in0 a)))
            nil)))

'((M1 (LOADINGS 2) (DRIVES 10)))
```

```
(setq ex-net12
      '((m1 (in0 in1 in2) (out0 out1 out2 out3)
            ((o1 (out1) id (in1))
             (o2 (out0) b-or (in0 out1))
             (o3 (out2) b-not (in1))
             (o4 (out3) b-and (in1 in2)))
            nil)))

 '((M1 (LOADINGS 1 (4 OUT1) 1)
       (DRIVES 10 IN1 10 10)))



(setq ex-net13
      '((m1 (in0 in1 in2) (out0 out1 out2 out3)
            ((o1 (out1) id (in1))
             (o2 (out0) b-or (in0 out1))
             (o3 (out2) b-not (out1))
             (o4 (out3) b-and (out1 in2)))
            nil)))

 '((M1 (LOADINGS 1 (4 OUT1) 1)
       (DRIVES 10 IN1 10 10)))



(setq ex-net14
      '((m1 (in0 clk) (out1 out2 out3)
            ((o1 (out1) b-not (in0))
             (o2 (is1) id (out1))
             (o3 (out2 w1) fd1 (is1 clk))
             (o4 (out3 w2) b-nbuf (is1)))
            o3)))

'((M1 (LOADINGS 2 1) (DRIVES 8 10 9)))



(setq ex-net15
      '((m1 (in1 in2) (out1 out2)
            ((o1 (is1) b-and (in1 in2))
             (o2 (out1) b-not-b4ip (is1))
             (o3 (out2) b-not (is1)))
            nil)))

'((M1 (LOADINGS 1 1) (DRIVES 40 10)))
```

```
(setq ex-net16
      '((m1 (in1 in2) (out1 out2 out3)
            ((o1 (out1) b-and (in1 in2))
             (o2 (out2) b-not-b4ip (out1))
             (o3 (out3) b-not (out1)))
            nil)))

'((M1 (LOADINGS 1 1) (DRIVES 0 40 10)))



(setq ex-net17
      '((m1 (in1 in2) (out1 out2)
            ((o1 (is1) b-and (in1 in2))
             (o2 (out1) b-not-b4ip (is1))
             (o3 (out2) b-not-ivap (is1)))
            nil)))

F



(setq ex-net18
      '((m1 (in1 in2) (out1 out2 out3)
            ((o1 (out1) b-and (in1 in2))
             (o2 (out2) b-not-b4ip (out1))
             (o3 (out3) b-not-ivap (out1)))
            nil)))

F



(setq ex-net19
      '((m1 (a b c d) (x)
            ((o1 (x) b-and (a b)))
            nil)))

'((M1 (LOADINGS 1 1 0 0) (DRIVES 10)))

|#
#|

(setq ex-net
      '((m1 (a) (b c d)
            ((o1 (b) id (a))
             (o2 (c) id (a))
             (o3 (d) id (a)))
            nil)))


(netlist-loadings-and-drives-simple ex-net)
```

```
(setq ex-net
      '((m1 (a) (b c d)
            ((o1 (b) id (a))
             (o2 (c) id (b))
             (o3 (d) id (c)))
            nil)))

(netlist-loadings-and-drives-simple ex-net)



(setq ex-net
      '((m1 (a) (b c d)
            ((o1 (b) b-not (a))
             (o2 (c) id (a))
             (o3 (d) id (a)))
            nil)))

(netlist-loadings-and-drives-simple ex-net)



(setq ex-net
      '((m1 (a) (b c d)
            ((o1 (b) b-not (a))
             (o2 (c) id (a))
             (o3 (d) id (b)))
            nil)))

(netlist-loadings-and-drives-simple ex-net)



(setq ex-net
      '((m1 (a) (b c d)
            ((o1 (b) b-not (a))
             (o2 (c) id (b))
             (o3 (d) id (c)))
            nil)))

(netlist-loadings-and-drives-simple ex-net)
```

```
(setq ex-net
      '((m1 (a) (c d)
            ((o1 (b) b-not (a))
             (o2 (c) id (b))
             (o3 (d) id (c)))
            nil)))


(netlist-loadings-and-drives-simple ex-net)



(setq ex-net '((m3 (in0 clk) (out1 out2)
                   ((o3 (is3 is4) fd1 (is1 is2))
                    (o1 (is1) id (in0))
                    (o2 (is2) id (clk))
                    (o4 (out1) id (is3))
                    (o5 (out2) id (is4)))
                   o3)))


(netlist-loadings-and-drives-simple ex-net)
 = '((M3 (LOADINGS 1 1) (DRIVES 10 10)))

|#
```

14.33 "predicate.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;
;;;    PREDICATE.EVENTS
;;;
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  Here we attempt to define a predicate that recognizes well-formed
;;;  circuits.  Syntactically, this is straightforward; however, for
;;;  bi-directional busses the ordering of primitives may result in
;;;  different evaluations.  There is also the concern about
;;;  combinational circularities in our circuit descriptions.
;;;  Hopefully, the predicates below do not admit such circuits.

;;;  The name of the predicate is TOP-LEVEL-PREDICATE.  It has one
;;;  parameter, a netlist (the normal HDL circuit description).

;;;  A second predicate LSI-TOP-LEVEL-PREDICATE has additional checks
;;;  for LSI Logic's requirements.  It has three parameters:
;;;    netlist - a netlist, in which all names are litatoms
;;;    token-size - maximum allowed name length (should be 64)
;;;    max-hierarchical-name-length - (should be 255)
;;;
;;;  The netlist predicates work in two stages.  First, they check the
;;;  netlist for syntax errors (functions NETLIST-SYNTAX-OK,
;;;  LSI-NETLIST-SYNTAX-OK).  Then, they check for other errors by
;;;  attempting to construct a database of netlist properties
;;;  (function NETLIST-PROPERTIES).

;;;  Also included is function NETLIST-DATABASE, which constructs a
;;;  database of netlist properties similar to the primitive database.
;;;  The netlist database does not contain LSI-NAME, NEW-STATES, and
;;;  RESULTS properties, but it does contain all the other primitive
;;;  properties.  NETLIST-DATABASE assumes there are no syntax errors
;;;  in the netlist.

;;;  Correct results from all the functions mentioned above depend on
;;;  a well-formed primitive database.  This is determined by function
;;;  PRIMP-DATABASE-PREDICATE (no parameters).

;;;  Test data for the predicates is in file predicate.tests.



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Syntax Requirements for Hardware Description Language
;;;
```

```
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;; The netlist syntax is:
;;;
;;;   <netlist> ::= ( <module>* )
;;;
;;;   <module> ::= ( <module name> <module inputs> <module outputs>
;;;                  <module occurrences> <module state names>
;;;                  [ <annotation> ] )
;;;
;;;   <module name> ::= <name>
;;;
;;;   <module inputs> ::= ( <name>* )
;;;
;;;   <module outputs> ::= ( <name>* )
;;;
;;;   <module occurrences> ::= ( <occurrence>+ )
;;;
;;;   <module state names> ::= <name> | ( <name> <name>+ )
;;;
;;;   <annotation> ::= ( ( <key> . <value> )* )
;;;
;;;   <occurrence> ::= ( <occurrence name> <occurrence outputs>
;;;                      <occurrence function> <occurrence inputs>
;;;                      [ <annotation> ] )
;;;
;;;   <occurrence name> ::= <name>
;;;
;;;   <occurrence outputs> ::= ( <name>* )
;;;
;;;   <occurrence function> ::= <name>
;;;
;;;   <occurrence inputs> ::= ( <name>* )
;;;
;;;   <name> ::= <simple name> | <indexed name>
;;;
;;;   <simple name> is a litatom
;;;
;;;   <indexed name> is a shell object, constructed by
;;;                  (index <simple name> <number>),
;;;                  where <number> is a numberp
;;;
;;;   <key> is unrestricted
;;;
;;;   <value> is unrestricted
;;;


;;;
;;; Additional syntax requirements are:
;;;
;;;   1.  There are no duplicate <module name>'s in the <netlist>, and no
;;;       <module name> is the same as a primitive name.
;;;
;;;   2.  There are no duplicate signal names within a <module> except for
```

```
;;;       IO-signals (intersection of <module inputs> and <module outputs>).
;;;       The signals of a module are <module inputs> and <occurrence outputs>
;;;       from all the <module occurrences>.
;;;
;;;   3.  All <module outputs> are generated once and only once as
;;;       <occurrence outputs> within the <module>.
;;;
;;;   4.  All <occurrence inputs> within a <module> are signals of the module.
;;;
;;;   5.  <Module state names> are a subset of the module's
;;;       <occurrence name>'s.
;;;
;;;   6.  There are no duplicate <occurrence name>'s within a module.
;;;
;;;   7.  Every <occurrence function> names either a primitive or a <module>
;;;       defined in the <netlist> before the <module> where the
;;;       <occurrence function> appears.
;;;
;;;   8.  Numbers of <occurrence inputs> and <occurrence outputs> are the same
;;;       as the numbers of inputs and outputs required by the definition of
;;;       the module or the description of the primitive named by the
;;;       <occurrence function>.


(defn name-okp (name)
  (if (litatom name)
      T
      (and (indexp name)
           (litatom (i-name name)))))


(disable name-okp)


(defn bad-names (name-list)
  (if (listp name-list)
      (let ((name   (car name-list))
            (result (bad-names (cdr name-list))))
        (if (name-okp name)
            result
            (insert name result)))
      nil))


(disable bad-names)


(defn name-list-errors (name-list duplicates-ok?)
  (list (nlistp-or-err (bad-names name-list) 'bad-names)
        (if duplicates-ok?
            T
            (no-duplicates-or-err name-list 'duplicates))
        (nil-or-err (last-cdr name-list) 'not-proper-list)))
```

```
(disable name-list-errors)


(defn name-list-ok (name-list label duplicates-ok?)
  (err-and (name-list-errors name-list duplicates-ok?)
           label))


(disable name-list-ok)


(defn occ-arg-length-error (fn expected-args actual-args label)
  (let ((f-length (length expected-args))
        (a-length (length actual-args)))
    (T-or-err (equal f-length a-length)
              label
              (list 'occ-function fn
                    (list 'expected f-length)
                    (list 'got      a-length)))))


(disable occ-arg-length-error)


(defn occ-form-ok (occ o-length)
  (err-and (list (T-or-err (or (equal 4 o-length)
                               (equal 5 o-length))
                           'bad-occurrence-length o-length)
                 (nil-or-err (last-cdr occ) 'not-proper-list))
           'occurrence-form))


(disable occ-form-ok)


(defn occ-name-ok (o-name o-length)
  (if (leq 1 o-length)
      (label-error (T-or-err (name-okp o-name) 'bad-name o-name)
                   'occurrence-name)
      T))


(disable occ-name-ok)


(defn occ-outputs-ok (o-outs o-length)
  (if (leq 2 o-length)
      (name-list-ok o-outs 'occurrence-outputs F)
      T))


(disable occ-outputs-ok)
```

```
(defn occ-inputs-ok (o-ins o-length)
  (if (leq 4 o-length)
      (name-list-ok o-ins 'occurrence-inputs T)
      T))


(disable occ-inputs-ok)



(defn occ-function-ok (o-fn o-ins o-outs o-length netlist)

  (cond ((lessp o-length 3)
         T)

        ((primp o-fn)
         (let ((f-ins  (primp2 o-fn 'inputs))
               (f-outs (primp2 o-fn 'outputs)))

           (err-and (list (if (leq 4 o-length)
                              (occ-arg-length-error o-fn f-ins o-ins
                                                    'wrong-number-of-inputs)
                              T)
                          (occ-arg-length-error o-fn f-outs o-outs
                                                'wrong-number-of-outputs))
                    'occurrence-function)))

        (T (let ((module (lookup-module o-fn netlist)))
             (if (listp module)
                 (let ((f-ins  (module-inputs  module))
                       (f-outs (module-outputs module)))

                   (err-and
                    (list (if (leq 4 o-length)
                              (occ-arg-length-error o-fn f-ins o-ins
                                                    'wrong-number-of-inputs)
                              T)
                          (occ-arg-length-error o-fn f-outs o-outs
                                                'wrong-number-of-outputs))
                    'occurrence-function))

                 (pred-error 'unknown-occurrence-function o-fn))))))


(disable occ-function-ok)


(defn occ-annotation-ok (o-anno o-length)
  (if (equal o-length 5)
      (label-error (T-or-err (alistp o-anno) 'not-alist o-anno)
                   'occurrence-annotation)
      T))
```

```
(disable occ-annotation-ok)


(defn occ-syntax-errors (occ netlist)
  ;; occ is an occurrence from a module

  (let ((o-name (occ-name       occ))
        (o-outs (occ-outputs    occ))
        (o-fn   (occ-function   occ))
        (o-ins  (occ-inputs     occ))
        (o-anno (occ-annotation occ))

        (o-length (length occ)))

    (list (occ-form-ok occ o-length)

          (occ-name-ok    o-name o-length)
          (occ-outputs-ok o-outs o-length)
          (occ-inputs-ok  o-ins  o-length)

          (occ-function-ok o-fn o-ins o-outs o-length netlist)

          (occ-annotation-ok o-anno o-length))))



(disable occ-syntax-errors)


(defn occ-syntax-ok (occ netlist) ; occ is a module occurrence
  (let ((o-name (occ-name occ)))
    (err-and (occ-syntax-errors occ netlist)
             (list 'occurrence o-name))))


(disable occ-syntax-ok)


;;; In OCC-BODY-SYNTAX-ERRORS, the parameters are as follows:
;;;    body - the list of occurrences from a module (or a tail of it).
;;;    occ-data - an alist containing:
;;;      signals - a list of defined signals, consisting of module inputs and
;;;                outputs of occurrences previous to body.
;;;      pending-IO - a list of IO signals that have not yet appeared as
;;;                   outputs.
;;;      non-depends - a list of signals that were not defined when they were
;;;                    used as inputs of occurrences previous to body.  The
;;;                    outputs of those occurrences should not have depended on
;;;                    the undefined inputs, but that check is made elsewhere.
;;;      occ-names - a list of the names of occurrences previous to body.
;;;    outputs - a list of the outputs of the module.
;;;    states - a list of the state names of the module.
;;;    netlist - the netlist
```

```
(defn initial-occ-syntax-data (m-ins m-outs)
  (let ((IO-signals (intersection m-ins m-outs)))
    (list (cons 'signals      m-ins)
          (cons 'pending-IO   IO-signals)
          (cons 'non-depends  nil)
          (cons 'occ-names    nil))))


(disable initial-occ-syntax-data)


(defn update-occ-syntax-data (occ occ-data)
  (let ((o-name (occ-name    occ))
        (o-outs (occ-outputs occ))
        (o-ins  (occ-inputs  occ))

        (signals      (value 'signals     occ-data))
        (pending-IO   (value 'pending-IO  occ-data))
        (non-depends  (value 'non-depends occ-data))
        (occ-names    (value 'occ-names   occ-data)))

    (list (cons 'signals     (append (if (listp pending-IO)
                                          (set-diff o-outs pending-IO)
                                          o-outs)
                                      signals))

          (cons 'pending-IO  (set-diff pending-IO o-outs))

          (cons 'non-depends (append (set-diff o-ins signals)
                                      non-depends))

          (cons 'occ-names   (cons o-name occ-names)))))


(disable update-occ-syntax-data)



(defn composite-occ-body-syntax-errors (body occ-data outputs states)
  (let ((signals     (value 'signals     occ-data))
        (pending-IO  (value 'pending-IO  occ-data))
        (non-depends (value 'non-depends occ-data))
        (occ-names   (value 'occ-names   occ-data)))

    (list (nil-or-err body 'not-proper-list)

          (nlistp-or-err pending-IO 'IO-signals-not-in-outputs)
          (subset-or-err outputs signals 'outputs-not-in-signals)
          (subset-or-err non-depends signals 'non-depends-not-in-signals)
          (subset-or-err states occ-names 'states-not-in-occ-names)

          (no-duplicates-or-err signals 'duplicates-in-signals)
          (no-duplicates-or-err occ-names 'duplicates-in-occ-names))))
```

```
(disable composite-occ-body-syntax-errors)


(defn occ-body-syntax-errors (body occ-data outputs states netlist)
  (if (nlistp body)
      (composite-occ-body-syntax-errors body occ-data outputs states)

      (let ((occ (car body)))

        (cons (occ-syntax-ok occ netlist)
              (occ-body-syntax-errors (cdr body)
                                      (update-occ-syntax-data occ occ-data)
                                      outputs states netlist)))))


(disable occ-body-syntax-errors)


(defn module-form-ok (module m-length)
  (err-and (list (T-or-err (or (equal m-length 5)
                               (equal m-length 6))
                           'bad-module-length m-length)
                 (nil-or-err (last-cdr module) 'not-proper-list))
           'module-form))


(disable module-form-ok)


(defn module-name-ok (m-name m-length netlist)
  (if (leq 1 m-length)
      (err-and (list (T-or-err (name-okp m-name) 'bad-name m-name)

                     (T-or-err (not (lookup-module m-name netlist))
                               'duplicate-module-defns m-name)

                     (T-or-err (not (primp m-name))
                               'module-name-same-as-primitive m-name))
               'module-name)
      T))


(disable module-name-ok)


(defn module-inputs-ok (m-ins m-length)
  (if (leq 2 m-length)
      (name-list-ok m-ins 'module-inputs F)
      T))


(disable module-inputs-ok)
```

```
(defn module-outputs-ok (m-outs m-length)
  (if (leq 3 m-length)
      (name-list-ok m-outs 'module-outputs F)
      T))


(disable module-outputs-ok)


(defn states-list-or-nil (m-states m-length)
  (if (leq 5 m-length)
      (m-states-list m-states)
      nil))


(disable states-list-or-nil)


(defn module-occurrences-ok (m-occs m-ins m-outs m-states m-length netlist)
  (if (leq 4 m-length)
      (err-and
       (cons (T-or-err (listp m-occs) 'no-occurrences m-occs)
             (occ-body-syntax-errors m-occs
                                     (initial-occ-syntax-data m-ins m-outs)
                                     m-outs
                                     (states-list-or-nil m-states m-length)
                                     netlist))
       'module-occurrences)
      T))


(disable module-occurrences-ok)


(defn module-statenames-ok (m-states m-length)
  (if (leq 5 m-length)
      (err-and (cons (T-or-err (if (listp m-states) (listp (cdr m-states)) T)
                               'list-of-length-1-not-allowed
                               m-states)
                     (name-list-errors (m-states-list m-states) F))
               'module-statenames)
      T))


(disable module-statenames-ok)


(defn module-annotation-ok (m-anno m-length)
  (if (equal m-length 6)
      (label-error (T-or-err (alistp m-anno) 'not-alist m-anno)
                   'module-annotation)
      T))
```

```
(disable module-annotation-ok)


(defn module-syntax-errors (module netlist)
  (let ((m-name   (module-name        module))
        (m-ins    (module-inputs      module))
        (m-outs   (module-outputs     module))
        (m-occs   (module-occurrences module))
        (m-states (module-statenames  module))
        (m-anno   (module-annotation  module))

        (m-length (length module)))

    (list (module-form-ok module m-length)

          (module-name-ok m-name m-length netlist)

          (module-inputs-ok  m-ins  m-length)
          (module-outputs-ok m-outs m-length)

          (module-occurrences-ok m-occs m-ins m-outs m-states m-length
                                 netlist)

          (module-statenames-ok m-states m-length)

          (module-annotation-ok m-anno   m-length))))


(disable module-syntax-errors)


(defn module-syntax-ok (module netlist)
  (let ((m-name (module-name module)))
    (err-and (module-syntax-errors module netlist)
             (list 'module m-name))))


(disable module-syntax-ok)


(defn netlist-syntax-errors (netlist)
  (if (listp netlist)
      (let ((module   (car netlist))
            (net-rest (cdr netlist)))
        (cons (module-syntax-ok module net-rest)
              (netlist-syntax-errors net-rest)))
      (list
       (nil-or-err netlist 'not-proper-list))))


(disable netlist-syntax-errors)
```

```
(defn netlist-syntax-ok (netlist)
  (err-and (netlist-syntax-errors netlist)
           'netlist-syntax-errors))


(disable netlist-syntax-ok)




;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Syntax Requirements for LSI Logic's Network Description Language
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  LSI Logic's Network Description Language (NDL) has the following
;;;  additional syntax requirements:
;;;
;;;  1.  A <name> is a standard litatom beginning with an upper case letter
;;;      and containing only upper case letters, digits, hyphens (-), and
;;;      underscores (_).  A <name> can contain at most 64 characters
;;;      (token-size).  No name can be an NDL keyword.
;;;
;;;  2.  LSI Logic uses hierarchical names to provide a unique name for each
;;;      signal in the circuit.  The hierarchical name is composed by appending
;;;      /<occurrence name> for each occurrence in an instance tree until a
;;;      primitive is reached; at that point /<name> is appended, where <name>
;;;      is the name of a signal in the occurrence that is an instance of the
;;;      primitive.  No hierarchical name can contain more than 255 characters
;;;      (hierarchical-size).
;;;
;;;  3.  No signal name (module input or occurrence output, including module
;;;      outputs) can be the same as an occurrence name except that if an
;;;      occurrence has only one output, then the output name can be the same
;;;      as the name of that occurrence.
;;;
;;;  4.  No signal name or occurrence name in a module can be the same as the
;;;      name of any function used within the module.  If the function
;;;      is a primitive, LSI Logic's name for it is the one that must
;;;      not be used as a signal or occurrence name.



(defn letterp (char)
  (member char
          (unpack 'ABCDEFGHIJKLMNOPQRSTUVWXYZ)))


(disable letterp)
```

```
(defn digitp (char)
  (member char
          (cdr (unpack 'a0123456789))))


(disable digitp)


(defn bad-lsi-name-chars (char-list special-characters)
  (if (listp char-list)
      (let ((char (car char-list))
            (rslt (bad-lsi-name-chars (cdr char-list) special-characters)))

        (if (or (letterp char)
                (digitp char)
                (member char special-characters))
            rslt
            (insert char rslt)))
      nil))


(disable bad-lsi-name-chars)



(defn lsi-keywords ()
  '(action angle annotations bloat block bus cell celstat chksum clk_line
    direction dummy end flip function hm_bidi hm_input hm_output library name
    nc net pin powercells priority psecured region scale technology tsecured
    via wire))


(disable lsi-keywords)

(disable *1*lsi-keywords)

;;; Following keywords cannot be used as named powers, as global signal names,
;;; in LCMP analyzed comment parameters, or in WIRED statements:
;;;
;;;     acpow acpower bd bidirect bidirects check compile dcpow dcpower
;;;     def define def_global def_globals del delay delays delete desc
;;;     description dir  directory filmgr in initialize input inputs level
;;;     list load mod module opt option options out output outputs power
;;;     powers probe security simsub str strength tab table tdl use
;;;     use_global use_globals wired
;;;
;;; also keywords:  #in #inp #input #inputs #out #output #outputs
```

```
(defn lsi-name-ok (name token-size)
  (cond ((member name (lsi-keywords))
         (pred-error 'lsi-name-is-keyword name))

        ((litatom name)
         (let ((unpacked-name (unpack name))
               (special-characters (cdr (unpack 'a-_))))
           (err-and
            (list (let ((nlength (length unpacked-name)))
                    (T-or-err (leq nlength token-size)
                              'name-too-long
                              (list (list 'name-length        nlength)
                                    (list 'max-allowed-length token-size))))

                  (T-or-err (letterp (car unpacked-name))
                            'name-begins-with-nonletter
                            (car unpacked-name))

                  (nlistp-or-err (bad-lsi-name-chars (cdr unpacked-name)
                                                     special-characters)
                                 'illegal-chars-in-name)

                  (let ((lc (last-cdr unpacked-name)))
                    (T-or-err (equal lc 0)
                              'not-standard-litatom
                              (list 'last-cdr lc))))

            (list 'bad-lsi-name name))))

      (T (pred-error 'lsi-name-not-litatom name))))


(disable lsi-name-ok)


(defn lsi-bad-names (name-list token-size)
  (if (listp name-list)
      (cons (lsi-name-ok   (car name-list) token-size)
            (lsi-bad-names (cdr name-list) token-size))
      nil))


(disable lsi-bad-names)



(defn lsi-name-list-errors (name-list token-size duplicates-ok?)
  (list (err-and (lsi-bad-names name-list token-size) 'bad-lsi-names)
        (if duplicates-ok?
            T
            (no-duplicates-or-err name-list 'duplicates))
        (nil-or-err (last-cdr name-list) 'not-proper-list)))
```

```
(disable lsi-name-list-errors)


(defn lsi-name-list-ok (name-list token-size label duplicates-ok?)
  (err-and (lsi-name-list-errors name-list token-size duplicates-ok?)
           label))


(disable lsi-name-list-ok)


(defn lsi-function-name (fname)
  (if (and (primp fname)
           (primp-lookup fname 'lsi-name))
      (let ((lsi-entry (primp2 fname 'lsi-name)))
        (if (listp lsi-entry)
            (car lsi-entry)
            lsi-entry))
      fname))


(disable lsi-function-name)



(defn name-length (name)
  (length (unpack name)))


(disable name-length)


(defn hierarchical-name-max (n1 n2)
  (cond ((and (nlistp n1) (nlistp n2))
         0)
        ((nlistp n1)
         n2)
        ((nlistp n2)
         n1)
        ((lessp (car n1) (car n2))
         n2)
        (T n1)))


(disable hierarchical-name-max)
```

```
(defn max-hierarchical-length-and-name (names)
  (if (listp names)
      (let ((n1   (car names))
            (rest (cdr names)))
        (hierarchical-name-max
           (list (add1 (name-length n1)) n1)
           (max-hierarchical-length-and-name rest)))
      0))


(disable max-hierarchical-length-and-name)



(defn max-occ-hierarchical-name (occ hierarchical-table hierarchical-size)
  (let ((o-name (occ-name     occ))
        (o-outs (occ-outputs  occ))
        (o-fn   (occ-function occ))
        (o-ins  (occ-inputs   occ)))
    (let ((base (if (primp o-fn)
                    (max-hierarchical-length-and-name (append o-ins o-outs))
                    (value o-fn hierarchical-table))))
      (if (listp base)
          (let ((length (plus (add1 (name-length o-name)) (car base)))
                (name   (cons o-name (cdr base))))
            (if (leq length hierarchical-size)
                (cons length name)
                (pred-error 'hierarchical-name-too-long
                            (list (list 'name-length length)
                                  (list 'max-allowed-length hierarchical-size)
                                  (list 'name name)))))
          0))))


(disable max-occ-hierarchical-name)


;; In LSI-OCC-SYNTAX-OK, the parameters are as follows:
;;    occ - an occurrence from a module.
;;    hierarchical-name-error - a net-errorp if any hierarchical name in occ
;;                              was too long.
;;    occ-data - an alist containing:
;;      signals - a list of defined signals, consisting of module inputs and
;;                outputs of previous occurrences.
;;      occ-names - a list of the names of previous occurrences.
;;    netlist - the netlist.
;;    token-size - the maximum number of characters allowed in a name.
```

```
(defn lsi-occ-name-ok (o-name o-length o-outs occ-data token-size)
  (if (leq 1 o-length)
      (let ((signals (value 'signals occ-data)))
        (err-and (list (lsi-name-ok o-name token-size)

                       (T-or-err (not (member o-name signals))
                                 'occ-name-is-signal o-name)

                       (T-or-err (or (equal (cdr o-outs) nil)
                                     (not (member o-name o-outs)))
                                 'occ-name-is-same-as-output-name o-name))

                 'occurrence-name))
      T))


(disable lsi-occ-name-ok)


(defn lsi-occ-outputs-ok (o-outs o-length occ-data token-size)
  (if (leq 2 o-length)
      (let ((occ-names (value 'occ-names occ-data)))
        (err-and (list (lsi-name-list-ok o-outs token-size 'occ-outputs F)
                       (disjoint-or-err o-outs occ-names
                                        'occ-outputs-in-occ-names))
                 'occurrence-outputs))
      T))


(disable lsi-occ-outputs-ok)



(defn lsi-occ-inputs-ok (o-ins o-length token-size)
  (if (leq 4 o-length)
      (lsi-name-list-ok o-ins token-size 'occurrence-inputs T)
      T))


(disable lsi-occ-inputs-ok)
```

```
(defn lsi-occ-syntax-ok (occ hierarchical-name-error occ-data netlist
                         token-size)

  (let ((o-name (occ-name       occ))
        (o-outs (occ-outputs    occ))
        (o-fn   (occ-function   occ))
        (o-ins  (occ-inputs     occ))
        (o-anno (occ-annotation occ))

        (o-length (length occ)))

    (err-and
     (list (occ-form-ok occ o-length)

           (lsi-occ-name-ok    o-name o-length o-outs occ-data token-size)
           (lsi-occ-outputs-ok o-outs o-length occ-data token-size)
           (lsi-occ-inputs-ok  o-ins  o-length token-size)

           (occ-function-ok o-fn o-ins o-outs o-length netlist)

           (occ-annotation-ok o-anno o-length)

           hierarchical-name-error)
     (list 'occurrence o-name))))
```

```
(disable lsi-occ-syntax-ok)


;;; In LSI-OCC-BODY-SYNTAX-CHECK, the parameters are as follows:
;;;    body - the list of occurrences from a module (or a tail of it).
;;;    occ-data - an alist containing:
;;;      signals - a list of defined signals, consisting of module inputs and
;;;                outputs of occurrences previous to body.
;;;      pending-IO - a list of IO signals that have not yet appeared as
;;;                   outputs.
;;;      non-depends - a list of signals that were not defined when they were
;;;                    used as inputs of occurrences previous to body.  The
;;;                    outputs of those occurrences should not have depended
;;;                    on the undefined inputs, but that check is made
;;;                    elsewhere.
;;;      occ-names - a list of the names of occurrences previous to body.
;;;      occ-fns - a list of the names of occurrence functions from
;;;                occurrences previous to body.  On primitives, the names
;;;                are the LSI function names from the primitives database.
;;;      hierarchical-name - biggest hierarchical name from previous
;;;                          occurrences, in form (<length> . <name>+).
;;;    outputs - a list of the outputs of the module.
;;;    states - a list of the state names of the module.
;;;    netlist - the netlist
;;;    token-size - the maximum number of characters allowed in a name.
;;;    hierarchical-table - table containing biggest hierarchical names for
;;;                         subordinate modules, in form (<length> . <name>+).
;;;    hierarchical-size - the maximum number of characters allowed in a
;;;                        hierarchical name.


(defn initial-lsi-occ-syntax-data (m-ins m-outs)
  (list* (cons 'occ-fns nil)
         (cons 'hierarchical-name 0)
         (initial-occ-syntax-data m-ins m-outs)))


(disable initial-lsi-occ-syntax-data)


(defn update-lsi-occ-syntax-data (occ o-hname occ-data)
  (let ((o-fn (occ-function occ))
        (occ-fns           (value 'occ-fns           occ-data))
        (hierarchical-name (value 'hierarchical-name occ-data)))

    (list* (cons 'occ-fns (cons (lsi-function-name o-fn) occ-fns))
           (cons 'hierarchical-name
                 (hierarchical-name-max hierarchical-name o-hname))
           (update-occ-syntax-data occ occ-data))))
```

```
(disable update-lsi-occ-syntax-data)


(defn lsi-occ-body-syntax-check (body occ-data outputs states netlist
                                  token-size
                                  hierarchical-table hierarchical-size)
  (if (nlistp body)
      (let ((signals   (value 'signals   occ-data))
            (occ-fns   (value 'occ-fns   occ-data))
            (occ-names (value 'occ-names occ-data)))

        (append (composite-occ-body-syntax-errors body occ-data outputs
                                                  states)

                (list (alist-entry 'hierarchical-name occ-data)

                      (disjoint-or-err signals   occ-fns
                                       'signal-occ-fn-overlap)
                      (disjoint-or-err occ-names occ-fns
                                       'occ-name-occ-fn-overlap))))

      (let ((occ (car body)))
        (let ((o-hname (max-occ-hierarchical-name occ hierarchical-table
                                                  hierarchical-size)))

          (cons (lsi-occ-syntax-ok occ o-hname occ-data netlist token-size)
                (lsi-occ-body-syntax-check
                   (cdr body)
                   (update-lsi-occ-syntax-data occ o-hname occ-data)
                   outputs states netlist token-size
                   hierarchical-table hierarchical-size))))))


(disable lsi-occ-body-syntax-check)


(defn lsi-module-name-ok (m-name m-length token-size netlist)
  (if (leq 1 m-length)
      (err-and (list (lsi-name-ok m-name token-size)

                     (T-or-err (not (lookup-module m-name netlist))
                               'duplicate-module-defns m-name)

                     (T-or-err (not (primp m-name))
                               'module-name-same-as-primitive m-name))
               'module-name)
      T))


(disable lsi-module-name-ok)
```

```
(defn lsi-module-inputs-ok (m-ins m-length token-size)
  (if (leq 2 m-length)
      (lsi-name-list-ok m-ins token-size 'module-inputs F)
      T))


(disable lsi-module-inputs-ok)


(defn lsi-module-outputs-ok (m-outs m-length token-size)
  (if (leq 3 m-length)
      (lsi-name-list-ok m-outs token-size 'module-outputs F)
      T))


(disable lsi-module-outputs-ok)



(defn lsi-module-occurrences-check (m-occs m-ins m-outs m-states m-length
                                    netlist token-size
                                    hierarchical-table hierarchical-size)
  (if (leq 4 m-length)
      (cons (T-or-err (listp m-occs) 'no-occurrences m-occs)
            (lsi-occ-body-syntax-check m-occs
                                       (initial-lsi-occ-syntax-data m-ins
                                                                    m-outs)
                                       m-outs
                                       (states-list-or-nil m-states m-length)
                                       netlist token-size
                                       hierarchical-table hierarchical-size))
      T))


(disable lsi-module-occurrences-check)
```

```
(defn lsi-module-syntax-check (module netlist token-size hierarchical-table
                               hierarchical-size)

  (let ((m-name   (module-name        module))
        (m-ins    (module-inputs      module))
        (m-outs   (module-outputs     module))
        (m-occs   (module-occurrences module))
        (m-states (module-statenames  module))
        (m-anno   (module-annotation  module))

        (m-length (length module)))

    (let ((occ-result (lsi-module-occurrences-check m-occs m-ins m-outs
                                                    m-states m-length
                                                    netlist token-size
                                                    hierarchical-table
                                                    hierarchical-size)))

        (list (cons m-name (value 'hierarchical-name occ-result))
              (err-and
               (list (module-form-ok module m-length)

                     (lsi-module-name-ok m-name m-length token-size netlist)

                     (lsi-module-inputs-ok  m-ins  m-length token-size)
                     (lsi-module-outputs-ok m-outs m-length token-size)

                     (err-and occ-result 'module-occurrences)

                     (module-statenames-ok m-states m-length)

                     (module-annotation-ok m-anno   m-length))

               (list 'module m-name))))))


(disable lsi-module-syntax-check)



(defn lsi-netlist-syntax-check (netlist token-size hierarchical-size)
  (if (listp netlist)
      (let ((table (lsi-netlist-syntax-check (cdr netlist) token-size
                                             hierarchical-size)))
        (append (lsi-module-syntax-check (car netlist) (cdr netlist)
                                         token-size table
                                         hierarchical-size)
                table))
      (list
       (nil-or-err netlist 'not-proper-list))))


(disable lsi-netlist-syntax-check)
```

```
(defn lsi-netlist-syntax-ok (netlist token-size hierarchical-size)
  (err-and (lsi-netlist-syntax-check netlist token-size hierarchical-size)
           'lsi-netlist-syntax-errors))


(disable lsi-netlist-syntax-ok)




;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Netlist Database
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;; The netlist database functions assume that the primitive database is ok,
;;; and that there are no syntax errors in the netlist.



(defn primitive-properties ()
  '(delays drives input-types inputs loadings lsi-name new-states
    out-depends output-types outputs results state-types states
    gates pads primitives transistors))


(disable primitive-properties)

(disable *1*primitive-properties)


(defn all-module-props ()
  '(delays drives input-types inputs loadings out-depends output-types
    outputs state-types states gates pads primitives transistors))


(disable all-module-props)

(disable *1*all-module-props)


;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;
;;    Netlist Database Utilities
;;
;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;; Representation for property value if it had an error or was undefined

(add-shell unknown nil unknownp ())
```

```
(defn value-or-unknown (key alist)
  (if (boundp key alist)
      (value key alist)
      (unknown)))


(disable value-or-unknown)


(defn collect-value-or-unknown (keys alist)
  (if (listp keys)
      (cons (value-or-unknown         (car keys) alist)
            (collect-value-or-unknown (cdr keys) alist))
      nil))


(disable collect-value-or-unknown)



;; Representatin of IO-signal output

(add-shell IO-out nil IO-outp
           ((IO-out-signal (none-of) false))) ; (one-of litatom indexp)


(defn mark-IO-out (name IO-outs)
  (if (member name IO-outs)
      (IO-out name)
      name))


(disable mark-IO-out)


(defn mark-IO-outs-0 (names IO-outs)
  (if (listp names)
      (cons (mark-IO-out    (car names) IO-outs)
            (mark-IO-outs-0 (cdr names) IO-outs))
      nil))


(disable mark-IO-outs-0)


(defn mark-IO-outs (names IO-outs)
  (if (disjoint IO-outs names)
      names
      (mark-IO-outs-0 names IO-outs)))


(disable mark-IO-outs)
```

```
(defn unmark-IO-out (name)
  (if (IO-outp name)
      (IO-out-signal name)
      name))

(disable unmark-IO-out)

(defn unmark-IO-outs (names)
  (if (listp names)
      (let ((n (unmark-IO-out  (car names)))
            (r (unmark-IO-outs (cdr names))))
        (if (and (equal n (car names))
                 (equal r (cdr names)))
            names
            (cons n r)))
      names))

(disable unmark-IO-outs)

(defn value2 (s in-map out-map)
  (if (boundp s in-map)
      (value s in-map)
      (value (unmark-IO-out s) out-map)))

(disable value2)

(defn collect-value2 (lst in-map out-map)
  (if (listp lst)
      (cons (value2         (car lst) in-map out-map)
            (collect-value2 (cdr lst) in-map out-map))
      nil))

(disable collect-value2)

(defn signal-namep (x)
  (or (litatom x) (indexp x) (IO-outp x)))

(disable signal-namep)
```

```
(defn parent-synonym0 (name slist used-names)
  (cond ((member name used-names) name)
        ((boundp name slist)
         (let ((v (value name slist)))
           (if (signal-namep v)
               (parent-synonym0 v slist (cons name used-names))
               name)))
        (T name))
  ((lessp (count (unbind-list used-names slist)))))

(disable parent-synonym0)

(defn parent-synonym (name slist)
  (parent-synonym0 name slist nil))

(disable parent-synonym)

(defn parent-synonyms-list (lst slist)
  (if (listp lst)
      (cons (parent-synonym       (car lst) slist)
            (parent-synonyms-list (cdr lst) slist))
      lst))

(disable parent-synonyms-list)

(defn extract-names (lst)
  (cond ((nlistp lst) nil)
        ((signal-namep (car lst))
         (insert (car lst) (extract-names (cdr lst))))
        (t (extract-names (cdr lst)))))
```

```
(disable extract-names)



;;;  EXTERNALIZE-PARENTS makes the parent synonym for all outputs that
;;;  are synonyms of an internal signal be the first output that is a
;;;  synonym for the internal signal.  For example, in the following,
;;;
;;;    (FANOUT-4 (A)
;;;             (Z0 Z1 Z2 Z3)
;;;             ((AA (AA) B-BUF (A))
;;;              (G0 (Z0) ID (AA))
;;;              (G1 (Z1) ID (AA))
;;;              (G2 (Z2) ID (AA))
;;;              (G3 (Z3) ID (AA)))
;;;             NIL)
;;;
;;;  Z0 gets the drive value of internal signal AA, and the other
;;;  outputs, Z1, Z2, and Z3, are marked as synonyms of Z0.


(defn externalize-parents (names alist inputs outputs)
  ;; guard: names is a subset of outputs

  (if (listp names)
      (let ((out1 (car names)))
        (let ((p1 (parent-synonym out1 alist)))

          (let ((new-alist
                 (if (or (member p1 inputs) (member p1 outputs))
                     alist
                     (let ((v (value-or-unknown p1 alist)))
                       (list* (cons p1 out1)
                              (cons out1 v)
                              (unbind out1 (unbind p1 alist)))))))

            (externalize-parents (cdr names) new-alist inputs outputs))))

      alist))
```

```
(disable externalize-parents)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Module Input/Output Types
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  Below are functions that check the types of module inputs and outputs.
;;;
;;;  Known input types:
;;;    (BOOLP CLK FREE LEVEL PARAMETRIC TRI-STATE TTL TTL-TRI-STATE)
;;;
;;;  Known output types:
;;;    ((<an input name>)
;;;     BOOLP CLK LEVEL PARAMETRIC TRI-STATE TTL TTL-TRI-STATE)

;;;  Contradictory type requirements give an error.  IO signals must
;;;  have tri-state output types and either tri-state or free input
;;;  types.  T-wire inputs must have tri-state input and output types.
;;;  Otherwise, the output type of a signal is required to be a
;;;  subtype of its input type.


;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;  Type Comparison Functions
;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn subtype (x y)
  (cond ((equal y 'free)
         t)

        ((equal x 'boolp)
         (member y '(boolp tri-state ttl-tri-state ttl)))

        ((equal x 'ttl-tri-state)
         (member y '(tri-state ttl-tri-state)))

        (t (equal x y))))


(disable subtype)
```

```
(defn types-compatiblep (t1 t2)
  (cond ((equal t1 t2)
         T)

        ((or (equal t1 'free) (equal t2 'free))
         T)

        ((equal t1 'tri-state)
         (equal t2 'ttl-tri-state))

        ((equal t2 'tri-state)
         (equal t1 'ttl-tri-state))

        (T F)))


(disable types-compatiblep)


(defn tri-state-typep (type)
  (member type
          '(tri-state ttl-tri-state)))


(disable tri-state-typep)



;;; ~~~~~~~~~~~~~~~~~~~~~~
;;;  Type Lookup Functions
;;; ~~~~~~~~~~~~~~~~~~~~~~


(defn type-value0 (n alist free used-names)
  (if (or (member n used-names)
          (not (boundp n alist)))

      (if free 'free (list n))

      (let ((type (value n alist)))

        (if (listp type)
            (type-value0 (car type) alist free
                         (cons n used-names))
            type)))

  ((lessp (count (unbind-list used-names alist)))))


(disable type-value0)
```

```
(defn type-value (n alist free)
  (type-value0 n alist free nil))


(disable type-value)


(defn input-type (name alist)
  (type-value (unmark-IO-out name) alist T))


(disable input-type)


(defn output-type (name out-map in-map)
  (let ((type (type-value name out-map F)))
    (if (listp type)
        (type-value (car type) in-map F)
        type)))


(disable output-type)



;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;  Updating Occurrence List Input and Output Types
;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn initial-occ-in-types ()
  nil)


(defn initial-occ-out-types ()
  nil)
```

```
(defn add-in-type (name type type-map)
  (let ((name (unmark-IO-out name)))

    (cond ((equal type 'free)
           type-map)

          ((boundp name type-map)
           (let ((old-type (value name type-map)))
             ;; (not (equal old-type 'free))

             (cond ((or (unknownp old-type)
                        (subtype old-type type))
                    type-map)

                   ((or (unknownp type)
                        (subtype type old-type))
                    (cons (cons name type)
                          (unbind name type-map)))

                   (T (list* (pred-error (list 'signal name)
                                         (list (list 'old-type old-type)
                                               (list 'new-type type)))
                             (cons name (unknown))
                             (unbind name type-map))))))

          (T (cons (cons name type) type-map)))))


(disable add-in-type)


(defn add-in-types (inputs types type-map)
  (if (listp inputs)
      (add-in-types (cdr inputs) (cdr types)
                    (add-in-type (car inputs) (car types)
                                 type-map))
      type-map))


(disable add-in-types)


(defn update-in-types (in-types o-ins o-fn-props)
  (let ((o-in-types (value 'input-types o-fn-props)))
    (add-in-types o-ins o-in-types in-types)))


(disable update-in-types)
```

```
(defn add-out-type (name type arg-map type-map)
  (cons (cons name
              (if (listp type)
                  (collect-value type arg-map)
                  type))
        type-map))


(disable add-out-type)


(defn add-out-types (outputs types arg-map type-map)
  (if (listp outputs)
      (add-out-types (cdr outputs) (cdr types) arg-map
                     (add-out-type (car outputs) (car types)
                                   arg-map type-map))
      type-map))


(disable add-out-types)


(defn update-out-types (out-types o-outs o-fn-props in-map)
  (let ((o-out-types (value 'output-types o-fn-props)))
    (add-out-types o-outs o-out-types in-map out-types)))


(disable update-out-types)



;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;  Composing Occurrence List Input and Output Types
;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn IO-type-error (name in-type out-type)
  (pred-error (list (if (IO-outp name) 'IO-signal 'signal)
                    (unmark-IO-out name))

              (list (list 'input-type  in-type)
                    (list 'output-type out-type))))


(disable IO-type-error)
```

```
(defn IO-types-compatible (name in-type out-type t-wire-ins)
  (cond ((unknownp in-type)
         (unknownp out-type))

        ((unknownp out-type)
         (unknownp in-type))

        ((IO-outp name)
         (if (and (tri-state-typep out-type)
                  (subtype out-type in-type))
             T
             (IO-type-error name in-type out-type)))

        ((member name t-wire-ins)
         (if (and (tri-state-typep in-type)
                  (types-compatiblep in-type out-type))
             T
             (IO-type-error name in-type out-type)))

        ((subtype out-type in-type)
         T)

        (T (IO-type-error name in-type out-type))))


(disable IO-types-compatible)


(defn ok-in-type (ok name out-type in-map)
  (cond ((or (not ok) (net-errorp ok))
         (set-value (unmark-IO-out name) (unknown) in-map))

        ((IO-outp name)
         (set-value (unmark-IO-out name) out-type in-map))

        (T in-map)))


(disable ok-in-type)


(defn ok-out-type (ok name out-map)
  (cond ((net-errorp ok)
         (cons ok (set-value name (unknown) out-map)))

        (ok
         out-map)

        (T (set-value name (unknown) out-map))))


(disable ok-out-type)
```

```
(defn transfer-in-type (name out-type in-map)
  (if (listp out-type)
      (add-in-type (car out-type)
                   (input-type name in-map)
                   in-map)
      in-map))


(disable transfer-in-type)


(defn compose-type (name type maps t-wire-ins)
  (let ((in-map  (car maps))
        (out-map (cdr maps)))

      (let ((in-type  (input-type name in-map))

            (out-type (if (listp type)
                          (output-type (car type) out-map in-map)
                          type)))

        (let ((ok (IO-types-compatible name in-type
                                       (if (listp out-type)
                                           'free
                                           out-type)
                                       t-wire-ins)))

          (cons (ok-in-type  ok name out-type in-map)
                (ok-out-type ok name out-map))))))


(disable compose-type)


(defn composed-type-maps (out-types in-map out-map t-wire-ins)
  (if (listp out-types)
      (let ((entry (car out-types)))

        (let ((name (car entry))
              (type (cdr entry)))

          (compose-type name type
                        (composed-type-maps (cdr out-types)
                                            (transfer-in-type name type
                                                              in-map)
                                            out-map t-wire-ins)
                        t-wire-ins)))

      (cons in-map out-map)))


(disable composed-type-maps)
```

```
(defn compose-IO-types (in-map out-map t-wire-ins)
  (composed-type-maps out-map in-map out-map t-wire-ins))


(disable compose-IO-types)



;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;  Collecting Module Input and Output Types
;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn collect-in-types (names alist)
  (if (listp names)
      (cons (type-value (car names) alist T)
            (collect-in-types (cdr names) alist))
      nil))


(disable collect-in-types)


(defn collect-out-type (name out-map inputs in-map)
  (let ((type (output-type name out-map in-map)))
    (if (and (listp type)
             (not (member (car type) inputs)))
        (unknown)                       ; error in out-depends
        type)))


(disable collect-out-type)


(defn collect-out-types (outputs out-map inputs in-map)
  (if (listp outputs)
      (cons (collect-out-type  (car outputs) out-map inputs in-map)
            (collect-out-types (cdr outputs) out-map inputs in-map))
      nil))


(disable collect-out-types)


(defn in-types-error (in-map)
  (err-and in-map 'input-type-conflicts))


(disable in-types-error)


(defn out-types-error (out-map)
  (err-and out-map 'IO-type-conflicts))
```

```
(disable out-types-error)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Module Loadings and Drives
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;; <loading> ::= <number> | (<number> pf)
;;;
;;; <drive> ::=   <number> | <name> | (<number> mA)
;;;             | ( UNKNOWN ) ;; There was an error.
;;;             | (MIN <drive> <drive>+)

;;; ~~~~~~~~~~~~~~~~~~~~~~
;;;  Recognizer Functions
;;; ~~~~~~~~~~~~~~~~~~~~~~


(defn mAp (dr)
  ;; (<number> [(point <number>)] mA)

  (case (length dr)
    (2 (and (numberp (car dr))
            (equal (cadr dr) 'mA)
            (equal (cddr dr) nil)))

    (3 (and (numberp (car dr))
            (let ((fr (cadr dr)))
              (and (equal (length fr) 2)
                   (equal (car fr) 'point)
                   (numberp (cadr fr))))
            (equal (caddr dr) 'mA)
            (equal (cdddr dr) nil)))

    (otherwise F)))


(disable mAp)


(defn pfp (ld)
  ;; (<number> pf)

  (if (equal (length ld) 2)
      (and (numberp (car ld))
           (equal (cadr ld) 'pf)
           (equal (cddr ld) nil))
      f))
```

```
(disable pfp)


;;; ~~~~~~~~~~~~~~~~~~~~~~
;;;  Conversion Functions
;;; ~~~~~~~~~~~~~~~~~~~~~~


(defn mA-to-pf (x)
  ; (guard (mAp x)) => x is (<number> [(point <number>)] mA)

  (list (plus (times (car x) 10)
              (cadadr x))
        'pf))


(disable mA-to-pf)


(defn pf-to-mA (x)
  ; (guard (pfp x)) => (<number> pf)

  (let ((r (remainder (car x) 10)))
    (if (zerop r)
        (list (quotient (car x) 10) 'mA)
        (list (quotient (car x) 10) (list 'point r) 'mA))))


(disable pf-to-mA)


(defn std-load-to-pf (x)
  (list (times x 10) 'pf))


(disable std-load-to-pf)


(defn pf-to-std-load (x)
  (let ((r (quotient (car x) 10)))

    (if (lessp (remainder (car x) 10) 5)
        r
        (plus r 1))))


(disable pf-to-std-load)


(defn mA-to-std-drive (x)
  (car x))
```

```
(disable mA-to-std-drive)


(defn std-drive-to-mA (x)
  (list x 'mA))


(disable std-drive-to-mA)



;;; ~~~~~~~~~~~~~~~~~~~~~~
;;;  Arithmetic Functions
;;; ~~~~~~~~~~~~~~~~~~~~~~


(defn zero-loadingp (a)
  (if (pfp a)
      (zerop (car a))
      (zerop a)))


(disable zero-loadingp)


(defn pf-plus (a b)
  ;; guard: (and (pfp a) (pfp b))
  (list (plus (car a) (car b)) 'pf))


(disable pf-plus)


(defn pf-difference (a b)
  ;; guard: (and (pfp a) (pfp b))
  (list (difference (car a) (car b)) 'pf))


(disable pf-difference)


(defn pf-lessp (a b)
  ;; guard: (and (pfp a) (pfp b))
  (lessp (car a) (car b)))


(disable pf-lessp)
```

```
(defn mA-lessp (a b)
  ;; guard: (and (mAp a) (mAp b))
  (cond ((lessp (car a) (car b)) t)
        ((equal (car a) (car b))
         (lessp (cadadr a) (cadadr b)))
        (t f)))


(disable mA-lessp)


(defn loading-plus (x y)
  (cond ((numberp x)
         (cond ((numberp y)
                (plus x y))
               ((pfp y)
                (pf-plus (std-load-to-pf x) y))
               (T x)))

        ((pfp x)
         (cond ((pfp y)
                (pf-plus x y))
               ((numberp y)
                (pf-plus x (std-load-to-pf y)))
               (T x)))

        ((or (numberp y) (pfp y))
         y)

        (T 0)))


(disable loading-plus)



;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;  Updating Occurrence List Loadings and Drives
;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;; <occurrence list LOADINGS property value>
;;;   ::= ( ( <occurrence input name> . <loading> )* )
;;;
;;; <occurrence list DRIVES property value>
;;;   ::= ( ( <occurrence output name> . <drive> )* )
;;;
;;; IO-signal output is represented as ( IO-OUT <signal name> ).


(defn initial-occ-loadings ()
  nil)
```

```
(defn initial-occ-drives ()
  nil)
```

```
(defn add-loading (name loading load-map)
  (if (zero-loadingp loading)
      load-map

      (if (boundp name load-map)
          (let ((old-loading (value name load-map)))
            (cons (cons name (loading-plus loading old-loading))
                  (unbind name load-map)))

          (cons (cons name loading) load-map))))
```

```
(disable add-loading)
```

```
(defn add-loadings (names loadings load-map)
  (if (listp names)
      (add-loadings (cdr names) (cdr loadings)
                    (add-loading (car names) (car loadings) load-map))
      load-map))
```

```
(disable add-loadings)
```

```
(defn update-loadings (loadings o-ins o-fn-props)
  (let ((o-loads (value 'loadings o-fn-props)))
    (add-loadings o-ins o-loads loadings)))
```

```
(disable update-loadings)
```

```
(defn local-drive (drive listp in-map out-map drive-map)
  (cond (listp
         (if (listp drive)
             (cons (local-drive (car drive) F in-map out-map drive-map)
                   (local-drive (cdr drive) T in-map out-map drive-map))
             nil))

        ((signal-namep drive)
         (let ((n (value2 drive in-map out-map)))
           (parent-synonym n drive-map)))

        ((and (listp drive) (equal (car drive) 'min))
         (cons 'min
               (local-drive (cdr drive) T in-map out-map drive-map)))

        (T drive)))


(disable local-drive)


(defn add-drives (outputs drives in-map out-map drive-map)
  (if (listp outputs)
      (let ((new-drive (cons (car outputs)
                             (local-drive (car drives) F
                                          in-map out-map drive-map))))
        (add-drives (cdr outputs) (cdr drives) in-map out-map
                    (cons new-drive drive-map)))
      drive-map))


(disable add-drives)


(defn update-drives (drives o-outs o-fn-props in-map out-map)
  (let ((o-drives (value 'drives o-fn-props)))
    (add-drives o-outs o-drives in-map out-map drives)))


(disable update-drives)



;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;  Composing Occurrence Loadings and Drives
;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
(defn transfer-loading (drive listp load loadings)
  (cond (listp (if (listp drive)
                   (transfer-loading
                     (cdr drive) T load
                     (transfer-loading (car drive) F load loadings))
                   loadings))

        ((signal-namep drive)
         (add-loading drive load loadings))

        ((and (listp drive) (equal (car drive) 'min))
         (transfer-loading (cdr drive) T load loadings))

        (T loadings)))


(disable transfer-loading)


;;; TRANSFER-LOADINGS copies loadings from signals to their parent synonyms.
;;; DRIVES is ordered so that if the drive value of any signal S depends on
;;; other signals, they all appear after S in DRIVES.


(defn transfer-loadings (loadings drives)
  (if (listp drives)
      (let ((e (car drives)))
      (let ((n (car e))
            (v (cdr e)))
        (transfer-loadings (transfer-loading v F (value n loadings) loadings)
                           (cdr drives))))

      loadings))


(disable transfer-loadings)



;;; NET-DRIVES subtracts loads from drive values, returning an error
;;; if a signal is overloaded.
;
```

```
(defn literal-net-drive (drive load label)
  (cond ((numberp drive)
         (let ((ld (cond ((numberp load)
                          load)
                         ((pfp load)
                          (pf-to-std-load load))
                         (T 0))))
           (if (lessp drive ld)
               (pred-error label
                           (list (list 'drive drive)
                                 (list* 'loading load
                                        (if (equal load ld) nil (list ld)))))
               (difference drive ld))))

        ((mAp drive)
         (let ((drpf (mA-to-pf drive))
               (ldpf (cond ((pfp load)
                            load)
                           ((numberp load)
                            (std-load-to-pf load))
                           (T '(0 pf)))))
           (if (pf-lessp drpf ldpf)
               (pred-error label
                           (list (list 'drive drive drpf)
                                 (list* 'loading load
                                        (if (equal load ldpf)
                                            nil
                                            (list ldpf)))))
               (pf-to-mA (pf-difference drpf ldpf)))))

        (T (unknown))))


(disable literal-net-drive)


(defn net-min-args (a load listp)
  (cond (listp (if (listp a)
                   (union (net-min-args (car a) load F)
                          (net-min-args (cdr a) load T))
                   nil))

        ((or (numberp a) (mAp a))
         (list (literal-net-drive a load 'min-arg)))

        ((and (listp a) (equal (car a) 'min))
         (net-min-args (cdr a) load T))

        (T (list a))))


(disable net-min-args)
```

```
(defn add-net-min-drive (name args load drive-map)
  (let ((args (net-min-args args load T)))

    (let ((err (err-and args (list 'signal name))))

      (if (net-errorp err)
          (let ((args (extract-names args)))
            (if (listp args)
                (list* err
                       (cons name
                             (cons 'min
                                   (cons (unknown) args)))
                       drive-map)
                (cons err drive-map)))

          (cons (cons name (cons 'min args))
                drive-map)))))


(disable add-net-min-drive)


(defn add-net-drive (drive loadings drive-map)
  (if (listp drive)
      (let ((n  (car drive))
            (v  (cdr drive)))

        (let ((ld (value n loadings)))

          (cond ((or (numberp v) (mAp v))
                 (let ((r (literal-net-drive v ld (list 'signal n))))
                   (cons (if (net-errorp r) r (cons n r))
                         drive-map)))

                ((equal (car v) 'min)
                 (add-net-min-drive n (cdr v) ld drive-map))

                (T (cons drive drive-map)))))

      (cons drive drive-map)))          ; an error


(disable add-net-drive)


(defn net-drives-simple (drs lds)
  (if (listp drs)
      (add-net-drive (car drs) lds
                     (net-drives-simple (cdr drs) lds))
      nil))
```

```
(disable net-drives-simple)


;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;  Collecting module loadings and drives
;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn type-loading (load type)
  (cond ((member type '(boolp clk level parametric tri-state))
         (if (pfp load) (pf-to-std-load load) load))

        ((member type '(ttl ttl-tri-state))
         (if (numberp load) (std-load-to-pf load) load))

        (T load)))


(disable type-loading)


(defn collect-loadings (names types loadings)
  (if (listp names)
      (cons (type-loading (value (car names) loadings) (car types))
            (collect-loadings (cdr names) (cdr types) loadings))
      nil))


(disable collect-loadings)


(defn loadings-error (loadings)
  (err-and loadings 'bad-loadings))


(disable loadings-error)
```

```
(defn collect-min-args (A listp drives used-names inputs outputs)
  (cond (listp
         (if (listp A)
             (union (collect-min-args (car A) F drives used-names
                                      inputs outputs)
                    (collect-min-args (cdr A) T drives used-names
                                      inputs outputs))
             nil))

        ((signal-namep A)
         (cond ((member A inputs)
                (list A))
               ((and (not (or (member A outputs)
                              (member A used-names)))
                     (boundp A drives))
                (collect-min-args (value A drives) F drives
                                  (cons A used-names) inputs outputs))
               (T (list (unknown)))))

        ((and (listp A) (equal (car A) 'min))
         (collect-min-args (cdr A) T drives used-names
                           inputs outputs))

        ((or (numberp A) (mAp A))
         (list A))

        (T (list (unknown))))

  ((ord-lessp (cons (add1 (count (unbind-list used-names drives)))
                    (count A)))))


(disable collect-min-args)



(defn numeric-drives (drs)
  (if (listp drs)
      (let ((d-1  (car drs))
            (rest (cdr drs)))

        (if (or (numberp d-1) (mAp d-1))
            (cons d-1 (numeric-drives rest))

            (numeric-drives rest)))
      nil))


(disable numeric-drives)
```

```
(defn type-drive (drive type)
  (cond ((member type '(boolp clk level parametric tri-state))
         (if (mAp drive) (mA-to-std-drive drive) drive))

        ((member type '(ttl ttl-tri-state))
         (if (numberp drive) (std-drive-to-mA drive) drive))

        (T drive)))


(disable type-drive)


(defn drive-lessp (x y)
  (cond ((and (numberp x) (numberp y))
         (lessp x y))

        ((and (mAp x) (mAp y))
         (mA-lessp x y))

        (t f)))


(disable drive-lessp)


(defn drive-min (lst type)
  (cond ((nlistp lst)
         (unknown))

        ((nlistp (cdr lst))
         (type-drive (car lst) type))

        (T
         (let ((r1 (type-drive (car lst) type))
               (r2 (drive-min (cdr lst) type)))

           (if (drive-lessp r1 r2)
               r1
               r2)))))


(disable drive-min)
```

```
(defn make-drive-min (args type drives inputs outputs)
  (let ((new-args (collect-min-args args T drives nil inputs outputs)))

    (let ((names   (extract-names  new-args))
          (numbers (numeric-drives new-args)))

      (cond ((nlistp names)
             (drive-min numbers type))

            ((nlistp numbers)
             (if (listp (cdr names))
                 (cons 'min names)
                 (list* 'min (unknown) names)))

            (t (let ((n (drive-min numbers type)))
                 (list* 'min n names)))))))


(disable make-drive-min)


(defn collect-drives-0 (names output-types drives inputs outputs)
  (if (listp names)
      (let ((v (value-or-unknown (car names) drives)))

        (cons (cond ((or (numberp v) (mAp v))
                     (type-drive v (car output-types)))

                    ((signal-namep v)
                     (parent-synonym v drives))

                    ((and (listp v) (equal (car v) 'min))
                     (make-drive-min (cdr v) (car output-types)
                                     drives inputs outputs))

                    (t (unknown)))

              (collect-drives-0 (cdr names) (cdr output-types)
                                drives inputs outputs)))
      nil))


(disable collect-drives-0)


(defn collect-drives (outputs output-types drives inputs)
  (collect-drives-0 outputs output-types
                    (externalize-parents outputs drives inputs outputs)
                    inputs outputs))


(disable collect-drives)
```

```
(defn drives-error (drives)
  (err-and drives 'bad-drives))
```

```
(disable drives-error)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Module Delays
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  Primitive delay:
;;;    <primitive delay> ::=
;;;         <input name>
;;;       | ( OR <input name> <input name> )
;;;       | ( ( <slope,LH> <pintercept,LH> )
;;;           ( <slope,HL> <pintercept,HL> ) )
;;;
;;;    <slope> ::= <number> | ( <number> PS-PF )
;;;
;;;    <pintercept> ::= <number>
;;;
;;;
;;;  Module delay:
;;;    <module delay> ::=   <input or output name>
;;;                       | ( UNKNOWN ) ;; There was an error.
;;;                       | ( OR <module delay> <module delay>+ )
;;;                       | ( ( <slope,LH> <mintercept,LH> )
;;;                           ( <slope,HL> <mintercept,HL> )
;;;                           <delay dependency>* )
;;;
;;;    <mintercept> ::= <number> | ( RANGE <number> <number> )
;;;
;;;    <delay dependency> ::=   <input or output name>
;;;                           | ( UNKNOWN ) ;; There was an error.
;;;                           | ( OR <delay dependency> <delay dependency>+ )
;;;                           | ( RANGE <number> <number> <delay dependency>* )
;;;                           | <number>
;;;
;;;  Occurrence data delay:
;;;    <occurrence data delay> ::= <module delay>
;;;
;;;  IO-signal output is represented as ( IO-OUT <signal name> ).


;;;  In the normal case, delay to produce a signal S is given by
;;;
;;;     ( ( <slope,LH> (range min-LH-intercept max-LH-intercept ) )
;;;       ( <slope,HL> (range min-HL-intercept max-HL-intercept ) )
;;;       <delay dependency>* )
;;;
;;;  where the <delay dependency>'s represent the amount of time
;;;  required to produce the inputs signal S depends on.  The <delay
;;;  dependency>'s normally have the form
```

```
;;;
;;;      (range min-input-delay max-input-delay <delay dependency>*)
;;;
;;;  When all <delay dependency>'s are completely determined, the
;;;  minimum and maximum amount of time to produce all the inputs is
;;;  added to the minimum and maximum intercept values of the S delay.
;;;
;;;  The problem with this approach is that many of the <delay
;;;  dependency>'s are not fully determined in a large netlist.  For
;;;  the CHIP-SYSTEM netlist, adding delays increased the size of the
;;;  database produced by NETLIST-DATABASE from about 500,000
;;;  characters to about 10,000,000 characters.
;;;
;;;  To reduce the volume of the delay representation, we introduce
;;;  the following approximation into the delay intercept and
;;;  dependency computations.
;;;
;;;     ((slope-LH (range LH-min LH-max))
;;;      (slope-HL (range HL-min HL-max))
;;;      (range min-in-1 max-in-1 <in-1-dep-1> <in-1-dep-2> ... <in-1-dep-m1>)
;;;      (range min-in-2 max-in-2 <in-2-dep-1> <in-2-dep-2> ... <in-2-dep-m2>)
;;;      ...
;;;      (range min-in-n max-in-n <in-n-dep-1> <in-n-dep-2> ... <in-n-dep-mn>)
;;;      <name-dep-1> <name-dep-2> ... <name-dep-n2>)
;;;  ~=
;;;    ((slope-LH (range (plus LH-min (max min-in-1 min-in-2 ... min-in-n))
;;;                      (plus LH-max (max max-in-1 max-in-2 ... max-in-n))))
;;;     (slope-HL (range (plus HL-min (max min-in-1 min-in-2 ... min-in-n))
;;;                      (plus HL-max (max max-in-1 max-in-2 ... max-in-n))))
;;;     <in-1-dep-1> <in-1-dep-2> ... <in-1-dep-m1>
;;;     <in-2-dep-1> <in-2-dep-2> ... <in-2-dep-m2>
;;;     ...
;;;     <in-n-dep-1> <in-n-dep-2> ... <in-n-dep-mn>
;;;     <name-dep-1> <name-dep-2> ... <name-dep-n2>)
;;;
;;;  The approximation tends to make both the minimum and the maximum
;;;  intercept values higher than they should be.  With the
;;;  approximation, the CHIP-SYSTEM database has 922,130 characters.


;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;  Extraction, Construction, and Recognizer Functions
;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn ps-pfp (x)
  (and (properp x)
       (equal (length x) 2)
       (numberp (car x))
       (equal (cadr x) 'ps-pf)))


(disable ps-pfp)
```

```
(defn delay-LH (d)
  ;; d is a <module delay>
  (if (and (listp d)
           (not (equal (car d) 'or)))
      (car d)
      F))


(disable delay-LH)


(defn delay-HL (d)
  ;; d is a <module delay>
  (if (and (listp d)
           (not (equal (car d) 'or)))
      (cadr d)
      F))


(disable delay-HL)


(defn delay-slope-LH (d)
  ;; d is a <module delay>
  (if (and (listp d)
           (not (equal (car d) 'or)))
      (caar d)
      F))


(disable delay-slope-LH)


(defn delay-slope-HL (d)
  ;; d is a <module delay>
  (if (and (listp d)
           (not (equal (car d) 'or)))
      (caadr d)
      F))


(disable delay-slope-HL)


(defn delay-intercept-LH (d)
  ;; d is a <module delay>
  (if (and (listp d)
           (not (equal (car d) 'or)))
      (cadar d)
      F))
```

```
(disable delay-intercept-LH)


(defn delay-intercept-HL (d)
  ;; d is a <module delay>
  (if (and (listp d)
           (not (equal (car d) 'or)))
      (cadadr d)
      F))


(disable delay-intercept-HL)


(defn delay-dependencies (d)
  ;; d is a <module delay>
  (if (and (listp d)
           (not (equal (car d) 'or)))
      (cddr d)
      nil))


(disable delay-dependencies)


(defn make-delay-0 (LH HL deps)
  (list* LH HL deps))


(disable make-delay-0)



(defn make-delay (s-LH i-LH s-HL i-HL deps)
  (make-delay-0 (list s-LH i-LH)
                (list s-HL i-HL)
                deps))


(disable make-delay)


(defn determined-delayp (d)
  ;; d is a <module delay>
  (and (listp d)
       (not (equal (car d) 'or))
       (nlistp (delay-dependencies d))))


(disable determined-delayp)
```

```
(defn make-range (lo hi deps)
  (if (and (equal lo hi)
           (nlistp deps))
      lo
      (list* 'range lo hi deps)))


(disable make-range)


(defn rangep (x)
  (if (listp x)
      (equal (car x) 'range)
      ;; (and (equal (car x) 'range)
      ;;      (listp (cddr x))
      ;;      (numberp (cadr x))
      ;;      (numberp (caddr x))
      ;;      (delay-dependenciesp (cdddr x)))
      F))


(disable rangep)


(defn range-min (x)
  (if (rangep x)
      (cadr x)
      x))


(disable range-min)


(defn range-max (x)
  (if (rangep x)
      (caddr x)
      x))


(disable range-max)


(defn range-dependencies (x)
  (if (rangep x)
      (cdddr x)
      nil))


(disable range-dependencies)
```

```
(defn determined-rangep (x)
  (if (rangep x)
      (nlistp (range-dependencies x))
      (numberp x)))


(disable determined-rangep)



;;; ~~~~~~~~~~~~~~~~~~
;;;  Delay arithmetic
;;; ~~~~~~~~~~~~~~~~~~


(defn per-std-load-delay-slope (slope)
  (if (ps-pfp slope)
      (times (car slope) 10)
      slope))


(disable per-std-load-delay-slope)


(defn per-pf-delay-slope (slope)
  (if (numberp slope)
      (let ((r (quotient slope 10)))
        (list (if (lessp (remainder slope 10) 5)
                  r
                  (add1 r))
              'ps-pf))
      slope))


(disable per-pf-delay-slope)


(defn range-plus (i1 i2 deps)
  (make-range (plus (range-min i1) (range-min i2))
              (plus (range-max i1) (range-max i2))
              deps))


(disable range-plus)
```

```
(defn slope-times-load (slope load)
  (cond ((numberp slope)
         (cond ((numberp load)
                (times slope load))
               ((pfp load)
                (times slope (pf-to-std-load load)))
               (T 0)))

        ((ps-pfp slope)
         (cond ((pfp load)
                (times (car slope) (car load)))
               ((numberp load)
                (times (car slope) (car (std-load-to-pf load))))
               (T 0)))

        (T 0)))


(disable slope-times-load)



;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;  Updating Occurrence List Delays
;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn initial-occ-delays ()
  nil)


(prove-lemma local-delay-count-help-0 (rewrite)
  (implies (listp x)
           (equal (count x)
                  (add1 (plus (count (car x)) (count (cdr x)))))))


(prove-lemma local-delay-count-help (rewrite)
  (implies (listp x)
           (equal (equal (count x) 0)
                  F)))


(disable local-delay-count-help-0)



(enable range-dependencies)

(enable delay-dependencies)
```

```
(defn local-delay (delay out-depends listp in-map out-map delay-map)
  (cond (listp
         (if (listp delay)
             (cons (local-delay (car delay) out-depends F
                                in-map out-map delay-map)
                   (local-delay (cdr delay) out-depends T
                                in-map out-map delay-map))
             nil))

        ((signal-namep delay)
         (parent-synonym (value2 delay in-map out-map)
                         delay-map))

        ((nlistp delay)
         delay)

        ((equal (car delay) 'or)
         (cons 'or
               (local-delay (cdr delay) F T in-map out-map delay-map)))

        ((rangep delay)
         (let ((rdeps (range-dependencies delay)))
           (if (listp rdeps)
               (make-range (range-min delay) (range-max delay)
                           (local-delay rdeps F T in-map out-map delay-map))
               delay)))

        (T;; regular slope-intercept delay
         (let ((deps (delay-dependencies delay)))
           (let ((rdeps (if (listp deps) deps out-depends)))
             (if (listp rdeps)
                 (make-delay-0 (delay-LH delay) (delay-HL delay)
                               (local-delay rdeps F T
                                            in-map out-map delay-map))
                 delay)))))

  ((ord-lessp (cons (add1 (count out-depends)) (count delay)))))



(disable local-delay-count-help)

(disable range-dependencies)

(disable delay-dependencies)

(disable local-delay)
```

```
(defn add-delays (outputs delays out-depends in-map out-map delay-map)
  (if (listp outputs)
      (let ((new-delay (cons (car outputs)
                             (local-delay (car delays) (car out-depends)
                                          F in-map out-map delay-map))))

        (add-delays (cdr outputs) (cdr delays) (cdr out-depends)
                    in-map out-map (cons new-delay delay-map)))
      delay-map))


(disable add-delays)


(defn update-delays (delays o-outs o-fn-props in-map out-map)
  (let ((o-delays      (value 'delays      o-fn-props))
        (o-out-depends (value 'out-depends o-fn-props)))

    (add-delays o-outs o-delays o-out-depends in-map out-map delays)))


(disable update-delays)



;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;  Composing Occurrence Delays
;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn add-load-delay (delay load)
  (if (zero-loadingp load)
      delay

      (let ((s-LH (delay-slope-LH     delay))
            (i-LH (delay-intercept-LH delay))
            (s-HL (delay-slope-HL     delay))
            (i-HL (delay-intercept-HL delay))
            (deps (delay-dependencies delay)))

        (make-delay s-LH
                    (range-plus i-LH (slope-times-load s-LH load) nil)
                    s-HL
                    (range-plus i-HL (slope-times-load s-HL load) nil)
                    deps))))


(disable add-load-delay)
```

```
(defn loaded-delay (delay load listp)
  (cond (listp
         (if (listp delay)
             (cons (loaded-delay (car delay) load F)
                   (loaded-delay (cdr delay) load T))
             nil))

        ((nlistp delay)
         delay)

        ((equal (car delay) 'or)
         (cons 'or (loaded-delay (cdr delay) load T)))

        (T;; regular slope-intercept delay
         (add-load-delay delay load))))


(disable loaded-delay)


(defn loaded-delays (delays loadings)
  (if (listp delays)
      (let ((entry  (car delays))
            (result (loaded-delays (cdr delays) loadings)))

        (if (nlistp entry)              ; an error
            (cons entry result)

            (let ((name  (car entry))
                  (value (cdr entry)))

              (let ((load (value name loadings)))

                (cons (cons name
                            (loaded-delay value load F))
                      result)))))

      delays))


(disable loaded-delays)



;;; ~~~~~~~~~~~~~~~~~~~~~~~~~
;;;  Collecting Module Delays
;;; ~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
(defn merge-input-delays (r1 r2 or-args?)
  ;; guard: (and (mergeable-input-delay-p r1 or-args? (x))
  ;;             (mergeable-input-delay-p r2 or-args? (x)))

  (cond ((or (rangep r1) (numberp r1))
         (if (or (rangep r2) (numberp r2))
             (make-range (if or-args?
                             (min (range-min r1) (range-min r2))
                           (max (range-min r1) (range-min r2)))

                         (max (range-max r1) (range-max r2))

                         (union (range-dependencies r1)
                                (range-dependencies r2)))

           (make-range (range-min r1) (range-max r1)
                       (insert r2 (range-dependencies r1)))))

        ((or (rangep r2) (numberp r2))
         (make-range (range-min r2) (range-max r2)
                     (insert r1 (range-dependencies r2))))

        (T (make-range 0 0 (insert r1 (list r2))))))


(disable merge-input-delays)


(defn mergeable-input-delay-p (d approxp)
  (if approxp
      (or (rangep d)
          (numberp d)
          (signal-namep d))
    (determined-rangep d)))


(disable mergeable-input-delay-p)
```

```
(defn insert-input-delay (d dlist or-args? approxp)
  ;; **** This function is a mess, unclear, and
  ;; **** does not weed out unnecessary values well.
  (cond ((unknownp d)
         dlist)

        ((nlistp dlist)
         (if (and or-args? (equal (car d) 'or))
             (cdr d)
           (list d)))

        ((and or-args? (listp d) (equal (car d) 'or))
         (insert-input-delay (cadr d)
                             (insert (car dlist)
                                     (union (cddr d) (cdr dlist)))
                             or-args? approxp))

        ((mergeable-input-delay-p d (and approxp (not or-args?)))
         (if (mergeable-input-delay-p (car dlist) (and approxp (not or-args?)))
             (cons (merge-input-delays d (car dlist) or-args?)
                   (cdr dlist))
           (cons d dlist)))

        ((mergeable-input-delay-p (car dlist) (and approxp (not or-args?)))
         (cons (car dlist)
               (insert d (cdr dlist))))

        (T (insert d dlist))))


(disable insert-input-delay)


(defn literal-delay-to-range (d)
  (let ((iLH (delay-intercept-LH d))
        (iHL (delay-intercept-HL d))
        (dep (delay-dependencies d)))

    (make-range (min (range-min iLH) (range-min iHL))
                (max (range-max iLH) (range-max iHL))
                dep)))


(disable literal-delay-to-range)
```

```
(defn or-delay-args-to-ranges (args result)
  (if (listp args)
      (let ((arg1 (car args))

    (new-result (or-delay-args-to-ranges (cdr args) result)))

        (cond ((nlistp arg1)
               (insert-input-delay arg1 new-result T F))

              ((equal (car arg1) 'or)
               (or-delay-args-to-ranges (cdr arg1) new-result))

              (T (insert-input-delay (literal-delay-to-range arg1)
                                     new-result T F))))

      result))


(disable or-delay-args-to-ranges)



(defn make-delay-or (args)
  (cond ((nlistp args)
         (unknown))

        ((listp (cdr args))
         (cons 'or args))

        ((signal-namep (car args))
         (list* 'or (unknown) args))

        (T (car args))))


(disable make-delay-or)


(defn delay-to-range (d)
  (cond ((nlistp d)
         d)

        ((equal (car d) 'or)
         (make-delay-or (or-delay-args-to-ranges (cdr d) nil)))

        (T (literal-delay-to-range d))))


(disable delay-to-range)



(enable range-dependencies)
```

```
(defn addable-input-delay (in-delays approxp)
  (if (and (listp in-delays)
           (nlistp (cdr in-delays)))
      (let ((d (car in-delays)))
        (if (if approxp
                (or (rangep d) (numberp d))
              (determined-rangep d))
            d
          F))
    F))
```

```
(disable addable-input-delay)
```

```
(defn input-delay0 (dep delays inputs outputs or-args? list? used-names
                    approxp)
  (cond (list?
         (if (listp dep)
             (insert-input-delay
              (input-delay0 (car dep) delays inputs outputs
                            or-args? F used-names approxp)
              (input-delay0 (cdr dep) delays inputs outputs
                            or-args? T used-names approxp)
              or-args? approxp)
           nil))

        ((nlistp dep)
         (cond ((numberp dep)
                dep)
               ((member dep inputs)
                dep)
               ((member dep used-names)
                (unknown))
               ((member dep outputs)
                (if or-args?
                    (unknown)
                  (parent-synonym dep delays)))
               ((boundp dep delays)
                (input-delay0 (delay-to-range (value dep delays))
                              delays inputs outputs or-args? F
                              (cons dep used-names) approxp))
               (T (unknown))))

        ((equal (car dep) 'or)
         (make-delay-or (input-delay0 (cdr dep) delays inputs outputs
                                      T T used-names approxp)))

        ((rangep dep)
         (let ((in-delays (input-delay0 (range-dependencies dep)
                                        delays inputs outputs
                                        F T used-names approxp)))
           (let ((in-delay (addable-input-delay in-delays approxp)))
             (if in-delay
                 (range-plus dep in-delay
                             (range-dependencies in-delay))
               (make-range (range-min dep) (range-max dep)
                           in-delays)))))

        (T (unknown)))

  ((ord-lessp (cons (add1 (count (unbind-list used-names delays)))
                    (count dep)))))


(disable range-dependencies)
```

```
(disable input-delay0)



(defn input-delay (dep delays inputs outputs or-args? list? used-names
                   approxp)
  (let ((r (input-delay0 dep delays inputs outputs or-args? list? used-names
                         approxp)))
    (let ((r1 (car r)))
      (if (and (listp r) (rangep r1))
          (if (and (equal (range-min r1) 0)
                   (equal (range-max r1) 0))
              (union (range-dependencies r1)
                     (cdr r))
            r)
        r))))


(disable input-delay)



(defn type-delay-slope (slope type)
  (cond ((member type '(boolp clk level parametric tri-state))
         (per-std-load-delay-slope slope))
        ((member type '(ttl ttl-tri-state))
         (per-pf-delay-slope slope))
        (T slope)))



(disable type-delay-slope)


(defn make-output-delay (delay type input-delay-list approxp)
  (let ((i-LH (delay-intercept-LH delay))
        (i-HL (delay-intercept-HL delay))
        (s-LH (type-delay-slope (delay-slope-LH delay) type))
        (s-HL (type-delay-slope (delay-slope-HL delay) type)))

    (let ((d-in (addable-input-delay input-delay-list approxp)))

      (if d-in

          (make-delay s-LH (range-plus i-LH d-in nil)
                      s-HL (range-plus i-HL d-in nil)
                      (range-dependencies d-in))

        (make-delay s-LH i-LH s-HL i-HL input-delay-list)))))


(disable make-output-delay)
```

```
(defn or-delay-args (A listp type delays used-names inputs outputs approxp)
  (cond (listp
         (if (listp A)
             (union (or-delay-args (car A) F type delays used-names
                                   inputs outputs approxp)
                    (or-delay-args (cdr A) T type delays used-names
                                   inputs outputs approxp))
             nil))

        ((signal-namep A)
         (cond ((member A inputs)
                (list A))
               ((and (not (or (member A outputs)
                              (member A used-names)))
                     (boundp A delays))
                (or-delay-args (value A delays) F type
                               delays (cons A used-names)
                               inputs outputs approxp))
               (T (list (unknown)))))

        ((nlistp A)
         (list (unknown)))

        ((equal (car A) 'or)
         (or-delay-args (cdr A) T type delays used-names
                        inputs outputs approxp))

        (T ;; regular slope-intercept-delay
         (list
          (make-output-delay A type
                             (input-delay (delay-dependencies A)
                                          delays inputs outputs
                                          F T used-names approxp)
                             approxp))))

  ((ord-lessp (cons (add1 (count (unbind-list used-names delays)))
                    (count A)))))


(disable or-delay-args)
```

```
(defn output-delay (name type delays inputs outputs approxp)
  (let ((v (value-or-unknown name delays)))

    (cond ((signal-namep v)
           (parent-synonym v delays))

          ((nlistp v)
           (unknown))

          ((equal (car v) 'or)
           (make-delay-or (or-delay-args (cdr v) T type delays (list name)
                                         inputs outputs approxp)))

          (T;; regular slope-intercept-delay
           (make-output-delay v type
                              (input-delay (delay-dependencies v)
                                           delays inputs outputs
                                           F T (list name) approxp)
                              approxp)))))


(disable output-delay)


(defn collect-delays-0 (names types delays inputs outputs approxp)
  (if (listp names)
      (cons (output-delay (car names) (car types) delays
                          inputs outputs approxp)
            (collect-delays-0 (cdr names) (cdr types) delays
                              inputs outputs approxp))
      nil))


(disable collect-delays-0)


(defn collect-delays (outputs output-types delays inputs approxp)
  (collect-delays-0 outputs output-types
                    (externalize-parents outputs delays inputs outputs)
                    inputs outputs approxp))


(disable collect-delays)


(defn delays-error (delays)
  (err-and delays 'bad-delays))
```

```
(disable delays-error)




;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Module Output Dependencies
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;; <occurrence list OUT-DEPENDS property value>
;;;   ::= ( <list of used signals>
;;;         ( <signal name> . <output-dependency>* )* )

;;;  An <output-dependency> is the name of a module input, on which
;;;  the signal depends.  It is an error if a used signal S
;;;  (occurrence input or module output) depends on an unknown signal
;;;  (including signals generated after S in the occurrence list).
;;;  The output dependency computation gives errors for circularities
;;;  (wiring loops or feedback) in circuit descriptions.



(defn initial-occ-out-depends (m-ins m-outs)
  (cons m-outs
        (pairlist m-ins (listify m-ins))))


(disable initial-occ-out-depends)


(defn add-out-depends (outputs out-depends arg-map deps-map)
  (if (listp outputs)
      (let ((out1 (car outputs))

            (od-1 (collect-value (car out-depends) arg-map)))

        (let ((new-entry (cons out1 (union-values od-1 deps-map)))
              (new-error (all-bound-or-err od-1 deps-map
                                           (list 'output out1))))

          (add-out-depends (cdr outputs) (cdr out-depends) arg-map
                           (list* new-entry new-error deps-map))))

      deps-map))


(disable add-out-depends)
```

```
(defn update-out-depends (out-depends o-outs o-fn-props o-ins arg-map)
  (let ((o-out-depends (value 'out-depends o-fn-props)))

    (cons (union o-ins (car out-depends))
          (add-out-depends o-outs o-out-depends arg-map (cdr out-depends)))))


(disable update-out-depends)




(defn collect-out-depends (outputs dependencies)
  (let ((map (cdr dependencies)))
    (collect-value outputs map)))


(disable collect-out-depends)


(defn collect-out-depends-errors (deps-map used-signals)
  (if (listp deps-map)
      (let ((entry  (car deps-map))

            (result (collect-out-depends-errors (cdr deps-map) used-signals)))

        (if (net-errorp entry)
            (let ((sname (cadr (error-label entry))))

              (if (member sname used-signals)
                  (cons entry result)

                  result))

            result))

      nil))


(disable collect-out-depends-errors)


(defn out-depends-error (dependencies)
  (let ((used-signals (car dependencies))
        (real-map     (cdr dependencies)))

    (nlistp-or-err (collect-out-depends-errors real-map used-signals)
                   'unbound-output-dependencies)))
```

```
(disable out-depends-error)




;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Module State Types
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  It is an error if the states listed in a module definition are
;;;  not the same as the states for which types are computed.



(defn initial-occ-state-types ()
  nil)


(defn update-state-types (state-types o-name o-fn-props)
  (if (boundp 'state-types o-fn-props)
      (cons (cons o-name
                  (value 'state-types o-fn-props))
            state-types)

      state-types))


(disable update-state-types)


(defn collect-state-types (states state-types)
  (if (or (listp states)
          (equal states nil))

      (collect-value-or-unknown states state-types)

      (value-or-unknown states state-types)))


(disable collect-state-types)


(defn state-types-error (state-types states)
  (let ((cstate-list (strip-cars state-types))
        (mstate-list (m-states-list states)))

    (T-or-err (set-equal cstate-list mstate-list)
              'bad-state-types
              (list (list 'module-states states)
                    (list 'computed-states cstate-list)))))
```

```
(disable state-types-error)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Module Tri-State Signal Checks
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;; 1.  A signal can be an input to at most one T-WIRE primitive.
;;;
;;; 2.  A signal that is an input to a T-WIRE primitive cannot be used for any
;;;     other purpose (occurrence input or module output).
;;;
;;; 3.  No IO-signal can be an input or an output of primitive ID (which
;;;     declares aliases for signal names).


;;; ~~~~~~~~~~~~~~~~~~~~~~~~~
;;;  Updating tri-state data
;;; ~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  Free signals are all module outputs and those occurrence inputs
;;;  that are not inputs to primitive t-wire.


(defn initial-occ-tri-state-data (m-outs)
  (list (cons 'free-signals m-outs)
        (cons 't-wire-ins   nil)
        (cons 'synonyms     nil)))


(disable initial-occ-tri-state-data)



(defn function-t-wire-ins (fn data)
  (cond ((boundp 'tri-state-data data)
         (value 't-wire-ins (value 'tri-state-data data)))

        ((equal fn 't-wire)
         (value 'inputs data))

        (T ;; data are the properties of a primitive
         nil)))


(disable function-t-wire-ins)
```

```
(defn function-synonyms (fn data)
  (cond ((boundp 'tri-state-data data)
         (value 'synonyms (value 'tri-state-data data)))

        ((equal fn 'id)
         (pairlist (value 'outputs data)
                   (value 'inputs  data)))

        (T ;; data are the properties of a primitive
         nil)))


(disable function-synonyms)



(defn add-synonyms (f-synonyms in-map out-map slist)
  (if (listp f-synonyms)
      (let ((s (car f-synonyms)))

        (add-synonyms (cdr f-synonyms) in-map out-map
                      (cons (cons (value  (car s) out-map)
                                  (value2 (cdr s) in-map out-map))
                            slist)))
      slist))


(disable add-synonyms)
```

```
(defn update-tri-state-data (data o-ins o-fn o-fn-data in-map out-map)
  (let ((old-free-signals (value 'free-signals data))
        (old-t-wire-ins   (value 't-wire-ins   data))
        (old-synonyms     (value 'synonyms     data)))

    (let ((occ-t-wire-ins (collect-value2 (function-t-wire-ins o-fn o-fn-data)
                                          in-map out-map)))

      (let ((occ-free-signals (cond ((listp occ-t-wire-ins)
                                     (set-diff o-ins occ-t-wire-ins))
                                    ((equal o-fn 'id)
                                     nil)
                                    (T o-ins)))

            (new-synonyms (add-synonyms (function-synonyms o-fn o-fn-data)
                                        in-map out-map old-synonyms)))

        (list (cons 'free-signals
                    (union occ-free-signals old-free-signals))

              (cons 't-wire-ins
                    (append occ-t-wire-ins old-t-wire-ins))

              (cons 'synonyms
                    new-synonyms))))))


(disable update-tri-state-data)



;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;  Composing tri-state data
;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn IO-signal-renames (synonyms IO-signals slist)
  (if (listp synonyms)
      (let ((e (car synonyms))
            (r (IO-signal-renames (cdr synonyms) IO-signals slist)))

      (let ((n (car e))
            (s (parent-synonym (cdr e) slist)))

        (if (and (or (IO-outp n) (IO-outp s)
                     (member s IO-signals))
                 (not (equal n s)))

            (cons (cons n s) r)

            r)))
      nil))
```

```
(disable IO-signal-renames)


(defn IO-rename-error (synonyms IO-signals)
  (nlistp-or-err (IO-signal-renames synonyms IO-signals synonyms)
                 'renamed-IO-signals))


(disable IO-rename-error)


(defn t-wire-error (free-signals t-wire-ins)
  (err-and (list (disjoint-or-err (remove-duplicates t-wire-ins)
                                  free-signals
                                  't-wire-ins-used-elsewhere)
                 (no-duplicates-or-err t-wire-ins
                                       'duplicate-t-wire-inputs))
           't-wire-errors))


(disable t-wire-error)


(defn compose-tri-state-data (data IO-signals)
  (let ((free-signals (value 'free-signals data))
        (t-wire-ins   (value 't-wire-ins   data))
        (synonyms     (value 'synonyms     data)))

    (let ((new-fs (parent-synonyms-list free-signals synonyms))
          (new-ts (parent-synonyms-list t-wire-ins   synonyms)))

      (list (cons 'free-signals new-fs)
            (cons 't-wire-ins   new-ts)
            (cons 'synonyms     synonyms)

            (IO-rename-error synonyms IO-signals)
            (t-wire-error new-fs new-ts)))))


(disable compose-tri-state-data)



;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;  Collecting tri-state data
;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
(defn collect-synonyms (m-outs synonyms)
  (if (listp m-outs)
      (let ((out1 (car m-outs))
            (rslt (collect-synonyms (cdr m-outs) synonyms)))

        (let ((p1 (parent-synonym out1 synonyms)))

          (if (equal out1 p1)
              rslt

              (cons (cons out1 p1) rslt))))

      nil))


(disable collect-synonyms)


(defn collect-tri-state-data (m-ins m-outs data)
  (let ((t-wire-ins (value 't-wire-ins data))
        (synonyms   (value 'synonyms   data)))

    (list (cons 't-wire-ins
                (intersection m-ins t-wire-ins))

          (cons 'synonyms
                (collect-synonyms m-outs
                                  (externalize-parents m-outs synonyms
                                                       m-ins m-outs))))))


(disable collect-tri-state-data)


(defn tri-state-errors (data)
  (let ((e1 (error-entry data 'renamed-IO-signals))
        (e2 (error-entry data 't-wire-errors)))

    (cond ((and (net-errorp e1) (net-errorp e2))
           (list e1 e2))

          ((net-errorp e1)
           e1)

          ((net-errorp e2)
           e2)

          (T nil))))
```

```
(disable tri-state-errors)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Collecting Occurrence Properties
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
(defn initial-occ-data (props m-ins m-outs)
  (if (listp props)
      (let ((prop1 (car props))

            (result1 (initial-occ-data (cdr props) m-ins m-outs)))

        (let ((entry1 (case prop1

                        (delays
                         (cons 'delays
                               (initial-occ-delays)))

                        (drives
                         (cons 'drives
                               (initial-occ-drives)))

                        (input-types
                         (cons 'input-types
                               (initial-occ-in-types)))

                        (loadings
                         (cons 'loadings
                               (initial-occ-loadings)))

                        (out-depends
                         (cons 'out-depends
                               (initial-occ-out-depends m-ins m-outs)))

                        (output-types
                         (cons 'output-types
                               (initial-occ-out-types)))

                        (state-types
                         (cons 'state-types
                               (initial-occ-state-types)))

                        (tri-state-data
                         (cons 'tri-state-data
                               (initial-occ-tri-state-data m-outs)))

                        (otherwise (if (member prop1
                                               '(gates pads primitives
                                                       transistors))
                                       (cons prop1 0)
                                       nil)))))
          (if (listp entry1)
              (cons entry1 result1)
              result1)))

      nil))
```

```
(disable initial-occ-data)



(defn update-oprop-binding (o-name o-ins o-outs o-fn o-fn-props in-map out-map
                            binding)
  (if (listp binding)
      (let ((p-name  (car binding))
            (p-value (cdr binding)))

        (cons p-name

              (case p-name
                (delays
                 (update-delays p-value o-outs o-fn-props in-map out-map))

                (drives
                 (update-drives p-value o-outs o-fn-props in-map out-map))

                (input-types
                 (update-in-types p-value o-ins o-fn-props))

                (loadings
                 (update-loadings p-value o-ins o-fn-props))

                (out-depends
                 (update-out-depends p-value o-outs o-fn-props o-ins in-map))

                (output-types
                 (update-out-types p-value o-outs o-fn-props in-map))

                (state-types
                 (update-state-types p-value o-name o-fn-props))

                (tri-state-data
                 (update-tri-state-data p-value o-ins o-fn o-fn-props
                                        in-map out-map))

                (otherwise
                 (if (member p-name '(gates pads primitives transistors))
                     (plus (value p-name o-fn-props) p-value)
                     p-value)))))

      binding))


(disable update-oprop-binding)
```

```
(defn update-occ-bindings (o-name o-ins o-outs o-fn o-fn-props in-map out-map
                           bindings)
  (if (listp bindings)
      (let ((prop-1 (car bindings))
            (b-rest (cdr bindings)))

        (cons (update-oprop-binding o-name o-ins o-outs o-fn o-fn-props
                                    in-map out-map prop-1)
              (update-occ-bindings  o-name o-ins o-outs o-fn o-fn-props
                                    in-map out-map b-rest)))

      nil))


(disable update-occ-bindings)




(defn function-properties (fn database)
  (if (primp fn)
      (cdr (primp fn))
      (value fn database)))


(disable function-properties)


(defn collect-occurrence-data (body bindings IO-outputs IO-signals database)
  (if (listp body)
      (let ((occ (car body)))

        (let ((o-name (occ-name     occ))
              (o-outs (occ-outputs  occ))
              (o-fn   (occ-function occ))
              (o-ins  (occ-inputs   occ)))

          (let ((o-fn-props (function-properties o-fn database))

                (o-ins2  (mark-IO-outs o-ins  IO-outputs))
                (o-outs2 (mark-IO-outs o-outs IO-signals)))

            (let ((in-map  (pairlist (value 'inputs  o-fn-props) o-ins2))
                  (out-map (pairlist (value 'outputs o-fn-props) o-outs2)))

              (collect-occurrence-data
               (cdr body)
               (update-occ-bindings o-name o-ins2 o-outs2 o-fn o-fn-props
                                    in-map out-map bindings)
               (append (intersection IO-signals o-outs) IO-outputs)
               IO-signals database)))))

      bindings))
```

```
(disable collect-occurrence-data)




;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Composing Occurrence Properties
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn composed-IO-types (in-types out-types tri-state-data IO-signals)
  (cons (cons 'tri-state-data
              (compose-tri-state-data tri-state-data IO-signals))

        (let ((types (compose-IO-types in-types out-types
                                       (value 't-wire-ins tri-state-data))))
          (list (cons 'input-types  (car types))
                (cons 'output-types (cdr types))))))


(disable composed-IO-types)


(defn composed-loadings-drives-delays (loadings drives delays)
  (let ((cloads (transfer-loadings loadings drives)))

    (list (cons 'delays
                (loaded-delays delays cloads))

          (cons 'drives
                (net-drives-simple drives cloads))

          (cons 'loadings cloads))))


(disable composed-loadings-drives-delays)
```

```
(defn add-composed-data (prop-name bindings IO-signals result)
  (cond ((or (boundp prop-name result)
             (not (boundp prop-name bindings)))
         result)

        ((member prop-name '(input-types output-types tri-state-data))
         (append (composed-IO-types (value 'input-types    bindings)
                                    (value 'output-types   bindings)
                                    (value 'tri-state-data bindings)
                                    IO-signals)
                 result))

        ((member prop-name '(loadings drives delays))
         (append (composed-loadings-drives-delays
                    (value 'loadings bindings)
                    (value 'drives   bindings)
                    (value 'delays   bindings))
                 result))

        (t (cons (alist-entry prop-name bindings)
                 result))))


(disable add-composed-data)



(defn composed-occurrence-data (bindings IO-signals result)
  (if (listp bindings)
      (let ((entry (car bindings)))

        (composed-occurrence-data (cdr bindings) IO-signals
                                  (if (nlistp entry) ; an error
                                      (cons entry result)
                                      (add-composed-data (car entry) bindings
                                                         IO-signals
                                                         result))))

      result))


(disable composed-occurrence-data)


(defn compose-occurrence-data (bindings IO-signals)
  (composed-occurrence-data bindings IO-signals nil))


(disable compose-occurrence-data)
```

```
(defn occurrence-data (props body m-ins m-outs IO-signals database)
  (let ((bindings (initial-occ-data props m-ins m-outs)))

    (compose-occurrence-data (collect-occurrence-data body bindings nil
                                                      IO-signals database)
                             IO-signals)))


(disable occurrence-data)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Collecting Module Properties
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
(defn collect-module-prop (prop-name m-ins in-types m-outs out-types m-states
                           obindings approximate-delays-p)
  (let ((entry (alist-entry prop-name obindings)))

    (let ((p-value (cdr entry)))

      (case prop-name

        (delays         (cons 'delays
                              (collect-delays m-outs out-types p-value
                                              m-ins approximate-delays-p)))

        (drives         (cons 'drives
                              (collect-drives m-outs out-types p-value
                                              m-ins)))

        (input-types    (cons 'input-types in-types))

        (inputs         (cons 'inputs m-ins))

        (loadings       (cons 'loadings
                              (collect-loadings m-ins in-types p-value)))

        (out-depends    (cons 'out-depends
                              (collect-out-depends m-outs p-value)))

        (output-types   (cons 'output-types out-types))

        (outputs        (cons 'outputs
                              (unmark-IO-outs m-outs)))

        (state-types    (if (equal m-states nil)
                            nil
                            (cons 'state-types
                                  (collect-state-types m-states p-value))))

        (states         (if (equal m-states nil)
                            nil
                            (cons 'states m-states)))

        (tri-state-data (cons 'tri-state-data
                              (collect-tri-state-data m-ins m-outs p-value)))

        (otherwise      entry)))))

(disable collect-module-prop)
```

```
(defn collect-module-props (props m-ins in-types m-outs out-types m-states
                            obindings approximate-delays-p)
  (if (listp props)
      (let ((prop1   (collect-module-prop  (car props) m-ins in-types
                                           m-outs out-types m-states
                                           obindings approximate-delays-p))

            (result1 (collect-module-props (cdr props) m-ins in-types
                                           m-outs out-types m-states
                                           obindings approximate-delays-p)))

        (if (listp prop1)
            (cons prop1 result1)
            result1))

      nil))


(disable collect-module-props)


(defn module-prop-error (obinding m-states)
  (if (listp obinding)
      (let ((p-name  (car obinding))
            (p-value (cdr obinding)))

        (case p-name
          (delays         (delays-error      p-value))
          (drives         (drives-error      p-value))
          (input-types    (in-types-error    p-value))
          (loadings       (loadings-error    p-value))
          (out-depends    (out-depends-error p-value))
          (output-types   (out-types-error   p-value))
          (state-types    (state-types-error p-value m-states))
          (tri-state-data (tri-state-errors  p-value))
          (otherwise      nil)))

      obinding))


(disable module-prop-error)
```

```
(defn collect-module-errors (obindings m-states)
  (if (listp obindings)
      (let ((error1 (module-prop-error     (car obindings) m-states))
            (errors (collect-module-errors (cdr obindings) m-states)))

        (cond ((net-errorp error1)
               (cons error1 errors))

              ((listp error1)
               (append error1 errors))

              (t errors)))

      nil))


(disable collect-module-errors)



(defn collect-module-data (props m-name m-ins m-outs m-states obindings
                           approximate-delays-p)
  (let ((m-in-types
         (if (member 'input-types props)
             (collect-in-types m-ins (value 'input-types obindings))
             nil))

        (m-out-types
         (if (member 'output-types props)
             (collect-out-types m-outs (value 'output-types obindings)
                                m-ins  (value 'input-types  obindings))
             nil)))

    (let ((m-data (collect-module-props props m-ins m-in-types m-outs
                                        m-out-types m-states obindings
                                        approximate-delays-p))

          (m-err  (nlistp-or-err (collect-module-errors obindings m-states)
                                 (list 'module m-name))))
      (if (net-errorp m-err)
          (list* m-name m-err m-data)
          (cons m-name m-data)))))


(disable collect-module-data)
```

```
(defn module-data (module props database approximate-delays-p)
  (let ((m-name   (module-name        module))
        (m-ins    (module-inputs      module))
        (m-outs   (module-outputs     module))
        (m-occs   (module-occurrences module))
        (m-states (module-statenames  module)))

    (let ((IO-signals (intersection m-ins m-outs)))

      (let ((m-outs (mark-IO-outs m-outs IO-signals)))

        (collect-module-data props m-name m-ins m-outs m-states
                             (occurrence-data props m-occs m-ins m-outs
                                              IO-signals database)
                             approximate-delays-p)))))


(disable module-data)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Constructing the Netlist Database
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn module-database (netlist props approximate-delays-p)
  ;; guard: (and (netlist-syntax-ok netlist)
  ;;             (set-equal (required-props props) props))
  (if (listp netlist)
      (let ((database (module-database (cdr netlist) props
                                       approximate-delays-p)))
        (cons (module-data (car netlist) props database
                           approximate-delays-p)
              database))
    nil))


(disable module-database)
```

```
(defn required-for-prop (prop)
  (case prop

    (delays
     '(delays loadings drives input-types output-types
       inputs outputs tri-state-data))

    (drives
     '(drives loadings input-types output-types
       inputs outputs tri-state-data))

    (input-types
     '(input-types output-types
       inputs outputs tri-state-data))

    (loadings
     '(loadings drives input-types output-types
       inputs outputs tri-state-data))

    (out-depends
     '(out-depends inputs outputs))

    (output-types
     '(output-types input-types
       inputs outputs tri-state-data))

    (otherwise
     (if (member prop (all-module-props))
         (list prop)
         nil))))


(disable required-for-prop)



(defn get-required-props (props wanted-props)
  (if (listp props)
      (union (required-for-prop  (car props))
             (get-required-props (cdr props) wanted-props))
      wanted-props))


(disable get-required-props)



(defn required-props (props)
  (get-required-props props props))
```

```
(disable required-props)

;(prove-lemma double-required-props nil
;  (set-equal (required-props (required-props p))
;             (required-props p)))



(defn unbind-netlist-props-0 (props database)
  (if (listp database)
      (let ((entry  (car database)))

        (let ((m-name  (car entry))
              (m-alist (cdr entry)))
        (let ((m-prop1 (car m-alist)))

          (cons (cons m-name
                      (if (net-errorp m-prop1)
                          (cons m-prop1
                                (unbind-list props (cdr m-alist)))
                          (unbind-list props m-alist)))

                (unbind-netlist-props-0 props
                                        (cdr database))))))

      nil))


(disable unbind-netlist-props-0)


(defn unbind-netlist-props (props database)
  (if (listp props)
      (unbind-netlist-props-0 (reverse props) database)
      database))


(disable unbind-netlist-props)
```

```
(defn netlist-properties (netlist props approximate-delays-p)
  ;; The predicate NETLIST-SYNTAX-OK should be used to ensure that the basic
  ;; syntax and arities of all modules in netlist are correct; the database
  ;; computation depends on this.  It also depends on the correctness of the
  ;; primitive database.
  ;; If APPROXIMATE-DELAYS-P is not F, the approximation described above
  ;; (under module delays) is applied when computing delays.

  (let ((props-check (subset-or-err props (all-module-props)
                                    'unknown-properties)))
    (if (net-errorp props-check)
        props-check

        (let ((rprops (required-props props)))

          (let ((database (module-database netlist rprops
                                           approximate-delays-p)))

            (let ((errs (collect-net-errors database)))

              (if (listp errs)
                  (pred-error 'netlist-errors errs)

                  (unbind-netlist-props (set-diff rprops props)
                                        database))))))))


(disable netlist-properties)



(defn netlist-database (netlist approximate-delays-p)
  ;; The predicate NETLIST-SYNTAX-OK should be used to ensure that the basic
  ;; syntax and arities of all modules in netlist are correct; the database
  ;; computation depends on this.  It also depends on the correctness of the
  ;; primitive database.
  ;; If APPROXIMATE-DELAYS-P is not F, the approximation described above
  ;; (under module delays) is applied when computing delays.

  (let ((props  (all-module-props)))
  (let ((rprops (required-props props)))

    (let ((database (module-database netlist rprops approximate-delays-p)))

      (unbind-netlist-props (set-diff rprops props)
                            database)))))
```

```
(disable netlist-database)




;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    The Predicates
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  RESULT:  A net-errorp is a false result; anything else is a true result.


(defn predicate-properties ()
  '(drives input-types loadings out-depends output-types state-types))


(disable predicate-properties)

(disable *1*predicate-properties)



(defn top-level-predicate (netlist)
  (let ((syntax-check (netlist-syntax-ok netlist)))
    (if (net-errorp syntax-check)
        syntax-check

        (let ((r (netlist-properties netlist (predicate-properties) T)))
          (if (net-errorp r)
              r
              T)))))


(disable top-level-predicate)



;;  LSI-TOP-LEVEL-PREDICATE parameters:
;;    netlist - a netlist, in which all names are litatoms
;;    token-size - maximum allowed name length (should be 64)
;;    max-hierarchical-name-length - (should be 255)
```

```
(defn lsi-top-level-predicate (netlist token-size
                               max-hierarchical-name-length)
  (let ((syntax-check
         (lsi-netlist-syntax-ok netlist token-size
                                max-hierarchical-name-length)))
    (if (net-errorp syntax-check)
        syntax-check

        (let ((r (netlist-properties netlist (predicate-properties) T)))
          (if (net-errorp r)
              r
              T)))))


(disable lsi-top-level-predicate)




;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    THE PRIMITIVE PREDICATE
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  The following functions check the primitive database for presence
;;;  and consistency of properties.  The primitive properties are:
;;;
;;;    '(DELAYS DRIVES INPUT-TYPES INPUTS LOADINGS LSI-NAME NEW-STATES
;;;      OUT-DEPENDS OUTPUT-TYPES OUTPUTS RESULTS STATE-TYPES STATES
;;;      GATES PADS PRIMITIVES TRANSISTORS)
;;;
;;;  PADS is not required, and the three state properties should
;;;  either all be present or all be absent.



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    PRIMITIVE PREDICATE UTILITIES
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn pname-listp (x)
  (if (listp x)
      (let ((n (car x))
            (r (cdr x)))
        (and (name-okp n)
             (not (member n r))
             (pname-listp r)))
      (equal x nil)))
```

```
(disable pname-listp)


(defn function-call-ok (x number-of-args)
  (if (and (listp x) (litatom (car x)))
      (if (properp x)
          (if (numberp number-of-args)
              (let ((ac (length (cdr x))))
                (T-or-err (equal ac number-of-args)
                          (list 'function (car x) 'has number-of-args 'args)
                          (list 'it 'is 'given ac 'in x)))

              T)
          (pred-error 'ill-formed-function-call x))
      (pred-error 'not-function-call x)))


(disable function-call-ok)



(defn IO-label (arg what)
  (if (numberp arg)
      (list 'unknown what arg)
      (list what arg)))


(disable IO-label)


(defn input-label (input)
  (IO-label input 'input))


(disable input-label)


(defn output-label (output)
  (IO-label output 'output))


(disable output-label)


(defn state-label (state)
  (IO-label state 'state))


(disable state-label)
```

```
(defn ucar (x)
  (if (listp x)
      (car x)
      (unknown)))


(disable ucar)


(defn ucdr (x)
  (if (listp x)
      (cdr x)
      (unknown)))


(disable ucdr)


(defn IO-car (args)
  (cond ((listp args) (car args))
        ((numberp args) args)
        (T 1)))


(disable IO-car)


(defn IO-cdr (args)
  (cond ((numberp args)
         (add1 args))
        ((nlistp args)
         2)
        ((numberp (cdr args))
         nil)
        (T (cdr args))))


(disable IO-cdr)



(defn out-prop-signals (out-depends inputs)
  (if (unknownp out-depends)
      inputs
      out-depends))


(disable out-prop-signals)
```

```
(defn id-out-prop-signals (output-type out-depends inputs)
  (if (listp output-type)
      output-type
      (out-prop-signals out-depends inputs)))


(disable id-out-prop-signals)




;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    PRIMITIVE INPUT AND OUTPUT TYPES
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;; Known input types:
;;; (BOOLP CLK FREE LEVEL PARAMETRIC TRI-STATE TTL TTL-TRI-STATE)
;;;
;;; Known output types:
;;; ((<an input name>) BOOLP CLK LEVEL PARAMETRIC TRI-STATE TTL TTL-TRI-STATE)



(defn all-input-types ()
  '(boolp clk free level parametric tri-state ttl ttl-tri-state))


(disable all-input-types)

(disable *1*all-input-types)


(defn all-output-types (inputs)
  (append (listify inputs)
          (delete* 'free (all-input-types))))


(disable all-output-types)
```

```
(defn pin-type-ok (type input IO-signals fname)
  (label-error (cond ((equal fname 't-wire)
                      (T-or-err (equal type 'tri-state)
                                't-wire-input-type-not-tri-state
                                type))

                     ((member input IO-signals)
                      (T-or-err (tri-state-typep type)
                                'bad-IO-signal-type
                                type))

                     (T (T-or-err (member type (all-input-types))
                                  'unknown-type
                                  type)))

               (input-label input)))


(disable pin-type-ok)


(defn pin-type-errors (types inputs IO-signals fname)
  (if (listp types)
      (list* (T-or-err (or (listp inputs) (numberp inputs))
                       'too-many-input-types
                       (list (length types) 'extras))
             (pin-type-ok     (car types) (IO-car inputs) IO-signals fname)
             (pin-type-errors (cdr types) (IO-cdr inputs) IO-signals fname))

      (list (nil-or-err types 'not-proper-list)
            (nlistp-or-err inputs 'inputs-without-types))))


(disable pin-type-errors)



(defn pin-types-ok (in-types alist)
  (let ((inputs  (value 'inputs     alist))
        (outputs (value 'outputs    alist))
        (fname   (value 'primp-name alist)))

    (let ((IO-signals (intersection inputs outputs))

          (inputs (if (unknownp inputs) 1 inputs)))

      (err-and (pin-type-errors in-types inputs IO-signals fname)
               'input-types))))


(disable pin-types-ok)
```

```
(defn pout-type-ok (type output out-depends inputs
                    IO-signals fname in-type-map)
  (label-error
   (cond ((equal fname 't-wire)
          (T-or-err (equal type 'tri-state)
                    't-wire-output-type-not-tri-state
                    type))

         ((member output IO-signals)
          (if (tri-state-typep type)
              (let ((in-type (value output in-type-map)))

                (T-or-err (or (not (tri-state-typep in-type))
                              (equal type in-type))
                          'different-IO-signal-in-and-out-types
                          (list (list 'input-type  in-type)
                                (list 'output-type type))))

              (pred-error 'bad-IO-signal-type type)))

         ((listp type)
          (let ((known (out-prop-signals out-depends inputs)))

            (T-or-err (and (properp type)
                           (equal (length type) 1)
                           (if (unknownp known)
                               (name-okp (car type))
                               (member (car type) known)))
                      'unknown-type type)))

         (T (T-or-err (member type (all-output-types nil))
                      'unknown-type
                      type)))

   (output-label output)))


(disable pout-type-ok)
```

```
(defn pout-type-errors (types outputs out-depends inputs IO-signals
                        fname in-type-map)
  (if (listp types)
      (list* (T-or-err (or (listp outputs) (numberp outputs))
                       'too-many-output-types
                       (list (length types) 'extras))

             (pout-type-ok (car types) (IO-car outputs) (ucar out-depends)
                           inputs IO-signals fname in-type-map)

             (pout-type-errors (cdr types) (IO-cdr outputs) (ucdr out-depends)
                               inputs IO-signals fname in-type-map))

      (list (nil-or-err    types   'not-proper-list)
            (nlistp-or-err outputs 'outputs-without-types))))


(disable pout-type-errors)


(defn pout-types-ok (out-types alist)
  (let ((outputs     (value 'outputs     alist))
        (out-depends (value 'out-depends alist))
        (inputs      (value 'inputs      alist))
        (in-types    (value 'input-types alist))
        (fname       (value 'primp-name  alist)))

    (let ((IO-signals  (intersection inputs outputs))
          (in-type-map (pairlist inputs in-types))

          (outputs (if (unknownp outputs) 1 outputs)))

      (err-and (pout-type-errors out-types outputs out-depends
                                 inputs IO-signals fname in-type-map)
               'output-types))))


(disable pout-types-ok)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    FUNCTIONS USED BY PRIMITIVE DELAYS AND DRIVES
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
(defn p-name-prop-value-ok (name out-type out-depends inputs label)
  (let ((known (id-out-prop-signals out-type out-depends inputs)))

    (T-or-err (if (unknownp known)
                  (name-okp name)
                  (member name known))
              label
              name)))


(disable p-name-prop-value-ok)


(defn p-t-wire-args-ok (args out-depends inputs label)
  (let ((known (out-prop-signals out-depends inputs)))

    (err-and (if (unknownp known)
                 (list (name-list-ok args 'ill-formed-args F))

                 (list (subset-or-err args known 'unknown-args)
                       (no-duplicates-or-err args 'duplicate-args)
                       (nil-or-err (last-cdr args) 'not-proper-list)))

             label)))


(disable p-t-wire-args-ok)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    PRIMITIVE DELAYS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
(defn phalf-delay-ok (d type label)
  (if (equal (length d) 2)
      (err-and (list (cond ((member type
                                    '(boolp clk level parametric tri-state))
                            (T-or-err (numberp (car d))
                                      'slope-not-number
                                      (list (list 'slope (car d))
                                            (list 'output-type type))))

                           ((member type '(ttl ttl-tri-state))
                            (T-or-err (ps-pfp (car d))
                                      'slope-not-ps-pf
                                      (list (list 'slope (car d))
                                            (list 'output-type type))))

                           (T (T-or-err (or (numberp (car d))
                                            (ps-pfp (car d)))
                                        'slope-not-number-or-ps-pf
                                        (car d))))

                     (T-or-err (numberp (cadr d))
                               'intercept-not-number (cadr d))

                     (nil-or-err (last-cdr d) 'not-proper-list))

               label)

      (pred-error label (list 'length-not-2 d))))


(disable phalf-delay-ok)
```

```
(defn pdelay-ok (delay output type out-depends inputs)
  (label-error
   (cond ((or (litatom delay) (indexp delay))
          (p-name-prop-value-ok delay type out-depends inputs
                                'unknown-delay))

         ((and (listp delay) (equal (car delay) 'or))
          (p-t-wire-args-ok (cdr delay) out-depends inputs
                            'bad-or-delay))

         ((equal (length delay) 2)
          (err-and
           (list (nlistp-or-err type 'expected-literal-type)
                 (phalf-delay-ok (car  delay) type 'low-to-high)
                 (phalf-delay-ok (cadr delay) type 'high-to-low)
                 (nil-or-err (last-cdr delay) 'not-proper-list))
           'bad-literal-delay))

         (T (pred-error 'unknown-delay delay)))

   (output-label output)))


(disable pdelay-ok)



(defn pdelays-errors (delays outputs types out-depends inputs)
  (if (listp delays)
      (list* (T-or-err (or (listp outputs) (numberp outputs))
                       'too-many-delays
                       (list (length delays) 'extras))

             (pdelay-ok (car delays) (IO-car outputs)
                        (ucar types) (ucar out-depends)
                        inputs)

             (pdelays-errors (cdr delays) (IO-cdr outputs)
                             (ucdr types) (ucdr out-depends)
                             inputs))

      (list (nil-or-err    delays  'not-proper-list)
            (nlistp-or-err outputs 'outputs-without-delays))))


(disable pdelays-errors)
```

```
(defn pdelays-ok (delays alist)
  (let ((outputs     (value 'outputs      alist))
        (out-types   (value 'output-types alist))
        (out-depends (value 'out-depends  alist))
        (inputs      (value 'inputs       alist)))

    (let ((outputs (if (unknownp outputs) 1 outputs)))

      (err-and (pdelays-errors delays outputs out-types out-depends inputs)
               'delays))))


(disable pdelays-ok)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    PRIMITIVE DRIVES
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn pdrive-ok (drive output type out-depends inputs)
  (label-error
   (cond ((or (litatom drive) (indexp drive))
          (p-name-prop-value-ok drive type out-depends inputs
                                'unknown-drive))

         ((and (listp drive) (equal (car drive) 'min))
          (p-t-wire-args-ok (cdr drive) out-depends inputs
                            'bad-min-drive))

         (T
          (let ((error-msg (list (list 'drive drive)
                                 (list 'type  type))))
            (cond ((member type
                           '(boolp clk level parametric tri-state))
                   (T-or-err (numberp drive) 'bad-drive error-msg))

                  ((member type '(ttl ttl-tri-state))
                   (T-or-err (mAp drive) 'bad-drive error-msg))

                  ((or (numberp drive) (mAp drive))
                   (T-or-err (nlistp type) 'expected-literal-type error-msg))

                  (T (pred-error 'unknown-drive drive))))))

   (output-label output)))


(disable pdrive-ok)
```

```
(defn pdrives-errors (drives outputs types out-depends inputs)
  (if (listp drives)
      (list* (T-or-err (or (listp outputs) (numberp outputs))
                       'too-many-drives
                       (list (length drives) 'extras))

             (pdrive-ok (car drives) (IO-car outputs)
                        (ucar types) (ucar out-depends)
                        inputs)

             (pdrives-errors (cdr drives) (IO-cdr outputs)
                             (ucdr types) (ucdr out-depends)
                             inputs))

      (list (nil-or-err    drives  'not-proper-list)
            (nlistp-or-err outputs 'outputs-without-drives))))


(disable pdrives-errors)


(defn pdrives-ok (drives alist)
  (let ((outputs     (value 'outputs      alist))
        (out-types   (value 'output-types alist))
        (out-depends (value 'out-depends  alist))
        (inputs      (value 'inputs       alist)))

    (let ((outputs (if (unknownp outputs) 1 outputs)))

      (err-and (pdrives-errors drives outputs out-types out-depends inputs)
               'drives))))


(disable pdrives-ok)



;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;
;;     PRIMITIVE LOADINGS
;;
;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
(defn ploading-ok (loading input type)
  (label-error
   (T-or-err (cond ((member type
                            '(boolp clk free level parametric tri-state))
                    (numberp loading))

                   ((member type '(ttl ttl-tri-state))
                    (pfp loading))

                   (t (or (numberp loading) (pfp loading))))

             'bad-loading

             (list (list 'loading loading)
                   (list 'input-type type)))

   (input-label input)))


(disable ploading-ok)


(defn ploadings-errors (loadings inputs types)
  (if (listp loadings)
      (list* (T-or-err (or (listp inputs) (numberp inputs))
                       'too-many-loadings
                       (list (length loadings) 'extras))

             (ploading-ok (car loadings) (IO-car inputs)
                          (ucar types))

             (ploadings-errors (cdr loadings) (IO-cdr inputs)
                               (ucdr types)))

      (list (nil-or-err    loadings 'not-proper-list)
            (nlistp-or-err inputs   'inputs-without-loadings))))


(disable ploadings-errors)


(defn ploadings-ok (loadings alist)
  (let ((inputs      (value 'inputs      alist))
        (input-types (value 'input-types alist)))

    (let ((inputs (if (unknownp inputs) 1 inputs)))

      (err-and (ploadings-errors loadings inputs input-types)
               'loadings))))
```

```
(disable ploadings-ok)


;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;     PRIMITIVE LSI-NAME ENTRY
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn plsi-name-ok (lsi-name-entry alist)
  (let ((inputs  (value 'inputs  alist))
        (outputs (value 'outputs alist))

        (lsi-name (if (listp lsi-name-entry)
                      (car lsi-name-entry)
                      lsi-name-entry)))

    (err-and (list (lsi-name-ok lsi-name 64)
                   ;; 64 is the max allowed lsi name length.
                   ;; It probably should be a parameter.

                   (if (or (nlistp lsi-name-entry) (unknownp inputs))
                       T
                       (T-or-err (set-equal (cdr lsi-name-entry) inputs)
                                 'input-conflict
                                 (list (list 'inputs inputs)
                                       (list 'lsi-inputs
                                             (cdr lsi-name-entry)))))

                   (if (or (unknownp inputs) (unknownp outputs))
                       T
                       (let ((IO-signals (intersection inputs outputs)))
                         (T-or-err (equal (listp IO-signals)
                                          (is-head (unpack 'bd)
                                                   (unpack lsi-name)))
                                   'bidirect-conflict
                                   (list (list 'lsi-name lsi-name)
                                         (list 'IO-signals IO-signals))))))

             'lsi-name)))


(disable plsi-name-ok)


;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;     PRIMITIVE OUT-DEPENDS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
(defn pout-depends-one-ok (out-depends output inputs)
  (if (unknownp inputs)
      (name-list-ok out-depends (output-label output) F)

      (err-and (list (subset-or-err out-depends inputs
                                    'unknown-output-dependencies)

                     (no-duplicates-or-err out-depends 'duplicates)

                     (nil-or-err (last-cdr out-depends) 'not-proper-list))

               (output-label output))))


(disable pout-depends-one-ok)


(defn pout-depends-errors (out-depends outputs inputs)
  (if (listp out-depends)
      (list* (T-or-err (or (listp outputs) (numberp outputs))
                       'too-many-out-depends
                       (list (length out-depends) 'extras))

             (pout-depends-one-ok (car out-depends) (IO-car outputs) inputs)

             (pout-depends-errors (cdr out-depends) (IO-cdr outputs) inputs))

      (list (nil-or-err    out-depends 'not-proper-list)
            (nlistp-or-err outputs     'outputs-without-out-depends))))


(disable pout-depends-errors)


(defn pout-depends-ok (out-depends alist)
  (let ((outputs (value 'outputs alist))
        (inputs  (value 'inputs  alist)))

    (let ((outputs (if (unknownp outputs) 1 outputs)))

      (err-and (pout-depends-errors out-depends outputs inputs)
               'out-depends))))


(disable pout-depends-ok)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    PRIMITIVE STATES
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
(defn pstates-ok (states)
  (cond ((listp states)
         (err-and (cons (T-or-err (listp (cdr states))
                                  'list-of-length-1-not-allowed
                                  states)
                        (name-list-errors states F))
                  'states))

        ((equal states nil)
         T)

        (T (T-or-err (name-okp states)
                     'states
                     (list 'bad-name states)))))


(disable pstates-ok)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;     PRIMITIVE STATE-TYPES
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


;;;  Function ADDRESSED-STATE is not called, but it is used as a type
;;;  descriptor in the primitive database.


(defn addressed-state (address-bits word-value)
  (if (zerop address-bits)
      word-value

      (cons (addressed-state (sub1 address-bits) word-value)
            (addressed-state (sub1 address-bits) word-value))))


(disable addressed-state)


(defn boolp-list-ok (lst label)
  (err-and (list (nlistp-or-err (remove-duplicates (delete* 'boolp lst))
                                'not-boolp)
                 (nil-or-err (last-cdr lst) 'not-proper-list))
           label))


(disable boolp-list-ok)
```

```
(defn literal-state-type-ok (type)
  (cond ((ramp type)
         (boolp-list-ok (ram-guts type) 'bad-ram-type))

        ((romp type)
         (boolp-list-ok (rom-guts type) 'bad-rom-type))

        ((stubp type)
         (boolp-list-ok (stub-guts type) 'bad-stub-type))

        (T
         (T-or-err (equal type 'boolp) 'unknown-type type))))


(disable literal-state-type-ok)



(defn pmemory-word-call-ok (type)
  (if (and (listp type)
           (member (car type) '(ram rom stub)))

      (let ((r (function-call-ok type 1)))

        (if (net-errorp r)
            r

            (let ((word (cadr type)))

              (label-error (if (equal (car word) 'quote)
                               (let ((r (function-call-ok word 1)))
                                 (if (net-errorp r)
                                     r
                                     (boolp-list-ok (cadr word)
                                                    'bad-memory-bit-types)))

                               (pred-error 'unknown-memory-bits-call word))

                           (list 'bad (car type) 'call)))))

      (pred-error 'not-memory-word-call type)))


(disable pmemory-word-call-ok)
```

```
(defn paddressed-state-call-ok (type)
  (if (and (listp type)
           (equal (car type) 'addressed-state))

      (let ((r (function-call-ok type 2)))
        (if (net-errorp r)
            r

            (err-and (list (T-or-err (numberp (cadr type))
                                     'address-bits-not-number
                                     (cadr type))
                           (pmemory-word-call-ok (caddr type)))
                     '(bad addressed-state call))))

      (pred-error 'not-addressed-state-call type)))


(disable paddressed-state-call-ok)


(defn one-pstate-type-ok (type)

  (cond ((nlistp type)
         (literal-state-type-ok type))

        ((member (car type) '(ram rom stub))
         (pmemory-word-call-ok type))

        ((equal (car type) 'addressed-state)
         (paddressed-state-call-ok type))

        (T (boolp-list-ok type 'bad-type-list))))


(disable one-pstate-type-ok)



(defn pstate-type-list-errors (types fname)
  (if (listp types)
      (let ((type (car types))
            (rest (cdr types)))

        (if (and (equal fname 'mem-32x32)
                 (member type '(numberp number-listp)))
            (pstate-type-list-errors rest fname)

            (insert (one-pstate-type-ok type)
                    (pstate-type-list-errors rest fname))))

      (list (nil-or-err types 'not-proper-list))))
```

```
(disable pstate-type-list-errors)


(defn pstate-type-ok (type state fname)
  (if (or (nlistp type)
          (member (car type) '(ram rom stub addressed-state)))

      (label-error (one-pstate-type-ok type)
                   (state-label state))

      (err-and (pstate-type-list-errors type fname)
               (state-label state))))


(disable pstate-type-ok)


(defn pstate-types-errors (types states fname)
  (if (listp types)
      (list* (T-or-err (or (listp states) (numberp states))
                       'too-many-state-types
                       (list (length types) 'extras))

             (pstate-type-ok      (car types) (IO-car states) fname)

             (pstate-types-errors (cdr types) (IO-cdr states) fname))

      (list (nil-or-err    types  'not-proper-list)
            (nlistp-or-err states 'states-without-types))))


(disable pstate-types-errors)


(defn pstate-types-ok (state-types alist)
  (let ((states (value 'states     alist))
        (fname  (value 'primp-name alist)))

    (err-and (if (or (listp states) (equal states nil))
                 (pstate-types-errors  state-types states fname)

                 (list (pstate-type-ok state-types states fname)))

             'state-types)))
```

```
(disable pstate-types-ok)


;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;     PRIMITIVE CHECKS FOR DUAL-EVAL FORMS (RESULTS, NEW-STATES)
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn unknown-args (x signals states arg-listp)
  (cond (arg-listp
         (if (listp x)
             (union (unknown-args (car x) signals states F)
                    (unknown-args (cdr x) signals states T))
             nil))

        ((nlistp x)
         (cond ((or (member x signals) (member x states))
                nil)
               ((or (unknownp signals) (unknownp states))
                (if (name-okp x) nil (list x)))
               (T (list x))))

        ((equal (car x) 'quote)
         nil)

        ((litatom (car x))
         (unknown-args (cdr x) signals states T))

        (T nil)))

(disable unknown-args)
```

```
(defn presult-form-errors (x arg-listp)
  (cond (arg-listp
         (cond ((listp x)
                (let ((r1 (presult-form-errors (car x) F))
                      (r2 (presult-form-errors (cdr x) T)))
                  (if (net-errorp r1)
                      (cons r1 r2)
                      (append r1 r2))))

               ((equal x nil)
                nil)

               (T (list (pred-error 'not-proper-list x)))))

        ((listp x)
         (let ((r (function-call-ok x (case (car x)
                                         (quote 1)
                                         (cons  2)
                                         (otherwise (unknown))))))
           (cond ((net-errorp r)
                  r)
                 ((equal (car x) 'quote)
                  nil)
                 (T (presult-form-errors (cdr x) T)))))

        (T nil)))


(disable presult-form-errors)



(defn presult-ok (result signals states label)
  (err-and (cons (nlistp-or-err (unknown-args result signals states F)
                                'unknown-args)

                 (presult-form-errors (list result) T))

           label))


(disable presult-ok)



(defn results-length (x)
  (cond ((nlistp x)
         0)
        ((equal (car x) 'cons)
         (add1 (results-length (caddr x))))
        (T 0)))
```

```
(disable results-length)



(defn presults-errors (results outputs out-depends inputs states label)
  (cond ((nlistp results)
         (list (pred-error 'unknown-dual-eval-form results)))

        ((equal results '(quote nil))
         (list (nlistp-or-err outputs (list 'missing label 'values))))

        ((equal (car results) 'cons)
         (let ((ok (function-call-ok results 2)))
           (if (net-errorp ok)
               (list ok)

               (let ((signals (out-prop-signals (ucar out-depends) inputs)))

                 (list* (T-or-err (or (listp outputs) (numberp outputs))
                                  'too-many-values
                                  (list (results-length results)
                                        'or 'more 'extras))

                        (presult-ok (cadr results) signals states
                                    (IO-label (IO-car outputs) label))

                        (presults-errors (caddr results) (IO-cdr outputs)
                                         (ucdr out-depends)
                                         inputs states label))))))

        ((or (listp outputs) (equal outputs nil))
         (list (pred-error (list 'evaluation 'should 'return
                                 (length outputs) 'results)
                           results)

               (presult-ok results inputs states
                           (list label 'list outputs))))

        (t (list (presult-ok results inputs states 'unknown-outputs)))))


(disable presults-errors)
```

```
(defn presults-ok (results alist)
  (let ((outputs     (value 'outputs     alist))
        (out-depends (value 'out-depends alist))
        (inputs      (value 'inputs      alist))
        (states      (value 'states      alist)))

    (let ((outputs (if (unknownp outputs) 1 outputs))
          (states  (if (unknownp states)
                       states
                       (m-states-list states))))

      (err-and (presults-errors results outputs out-depends
                                inputs states 'output)
               'results))))


(disable presults-ok)



(defn pnew-states-ok (new-states alist)
  (let ((states (value 'states alist))
        (inputs (value 'inputs alist)))

    (err-and (cond ((or (listp states) (equal states nil))
                    (presults-errors new-states states (unknown)
                                     inputs states 'state))

                   ((or (litatom states) (indexp states))
                    (list (presult-ok new-states inputs (list states)
                                      (state-label states))))

                   (T (presults-errors new-states 1 (unknown)
                                       inputs (unknown) 'state)))

             'new-states)))


(disable pnew-states-ok)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;     CHECK OF ONE PRIMITIVE DATABASE ENTRY
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
(defn primitive-prop-ok (prop alist keys)
  (if (listp prop)
      (let ((pname  (car prop))
            (pvalue (cdr prop)))

        (if (member pname keys)
            (pred-error 'duplicate-property prop)

            (case pname
              (delays       (pdelays-ok      pvalue alist))
              (drives       (pdrives-ok      pvalue alist))
              (input-types  (pin-types-ok    pvalue alist))
              (inputs       (name-list-ok    pvalue 'inputs F))
              (loadings     (ploadings-ok    pvalue alist))
              (lsi-name     (plsi-name-ok    pvalue alist))
              (new-states   (pnew-states-ok  pvalue alist))
              (out-depends  (pout-depends-ok pvalue alist))
              (output-types (pout-types-ok   pvalue alist))
              (outputs      (name-list-ok    pvalue 'outputs F))
              (results      (presults-ok     pvalue alist))
              (state-types  (pstate-types-ok pvalue alist))
              (states       (pstates-ok      pvalue))
              (otherwise    (if (member pname
                                        '(gates pads primitives transistors))
                                (T-or-err (numberp pvalue) pname
                                          (list 'value-not-number pvalue))
                                (pred-error 'unknown-property pname))))))

      (pred-error 'not-key-value-pair prop)))


(disable primitive-prop-ok)


(defn primitive-prop-errors (props alist keys)
  (if (listp props)
      (let ((prop (car props))
            (rest (cdr props)))

        (cons (primitive-prop-ok prop alist keys)
              (primitive-prop-errors rest alist (cons (car prop) keys))))

      (let ((mps (set-diff (primitive-properties) keys))
            (sps '(new-states state-types states)))

        (let ((req-mps (delete* 'pads
                                (if (subset sps mps)
                                    (set-diff mps sps)
                                    mps))))

          (list (nlistp-or-err req-mps 'missing-properties)
                (nil-or-err    props   'not-proper-list))))))
```

```
(disable primitive-prop-errors)



(defn ok-name-list (pname props)
  (let ((pvalue (value-or-unknown pname props)))

    (if (pname-listp pvalue)
        pvalue
        (unknown))))


(disable ok-name-list)


(defn ok-states (props)
  (if (boundp 'states props)
      (let ((states (value 'states props)))

        (cond ((or (listp states) (equal states nil))
               (if (pname-listp states)
                   states
                   (unknown)))

              ((name-okp states)
               states)

              (T (unknown))))

      nil))


(disable ok-states)


(defn ok-out-depends (ods inputs)
  (if (listp ods)
      (let ((od1 (car ods)))

        (cons (if (and (pname-listp od1)
                       (or (unknownp inputs)
                           (subset od1 inputs)))
                  od1
                  (unknown))

              (ok-out-depends (cdr ods) inputs)))

      nil))


(disable ok-out-depends)
```

```
(defn ok-input-types (types)
  (if (listp types)
      (let ((t1 (car types)))

        (cons (if (member t1 (all-input-types))
                  t1
                  (unknown))

              (ok-input-types (cdr types))))

      nil))


(disable ok-input-types)



(defn ok-output-types (out-types out-depends inputs)
  (if (listp out-types)
      (let ((t1 (car out-types)))

        (let ((ok (if (listp t1)
                      (let ((names (out-prop-signals (ucar out-depends)
                                                     inputs)))
                        (and (properp t1)
                             (equal (length t1) 1)
                             (if (unknownp names)
                                 (name-okp (car t1))
                                 (member (car t1) names))))

                      (member t1 (all-output-types nil)))))

          (cons (if ok t1 (unknown))
                (ok-output-types (cdr out-types) (ucdr out-depends) inputs))))

      nil))


(disable ok-output-types)
```

```
(defn parent-props-alist (primp-entry)
  (let ((pname (car primp-entry))
        (props (cdr primp-entry)))

    (let ((inputs  (ok-name-list 'inputs  props))
          (outputs (ok-name-list 'outputs props))

          (states  (ok-states props)))

      (let ((out-depends (ok-out-depends (value 'out-depends props) inputs)))

        (let ((input-types  (ok-input-types  (value 'input-types  props)))
              (output-types (ok-output-types (value 'output-types props)
                                             out-depends inputs)))

          (list (cons 'primp-name   pname)
                (cons 'inputs       inputs)
                (cons 'outputs      outputs)
                (cons 'states       states)
                (cons 'out-depends  out-depends)
                (cons 'input-types  input-types)
                (cons 'output-types output-types)))))))


(disable parent-props-alist)


(defn primitive-ok (p)
  (if (listp p)
      (let ((pname (car p))
            (props (cdr p)))

        (err-and (cons (T-or-err (name-okp pname) 'bad-primitive-name pname)
                       (primitive-prop-errors props
                                              (parent-props-alist p)
                                              nil))
                 (list 'primitive pname)))

      (pred-error 'not-a-primitive p)))


(disable primitive-ok)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;     PRIMITIVE PREDICATE PROPER
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
(defn primp-database-errors (d)
  (if (nlistp d)
      nil
      (cons (primitive-ok (car d))
            (primp-database-errors (cdr d)))))


(disable primp-database-errors)


(defn primp-database-predicate ()
  (err-and (primp-database-errors (primp-database))
           'primp-database-errors))


(disable primp-database-predicate)



;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    Functions that are not part of the predicate
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;; ~~~~~~~~~~~
;;;  Utilities
;;; ~~~~~~~~~~~


(enable boundp)
```

```
(defn subnet0 (flg x0 netlist used-modules result)
  (case flg
    (0 (let ((mname x0))
         (if (or (lookup-module mname result)
                 (member mname used-modules))
             result
             (let ((module (lookup-module mname netlist)))
               (if (listp module)
                   (let ((r (subnet0 1 (module-occurrences module)
                                     netlist (cons mname used-modules)
                                     result)))
                     (cons module r))
                   result)))))
    (1 (let ((occs x0))
         (if (nlistp occs)
             result
             (subnet0 1 (cdr occs) netlist used-modules
                      (subnet0 0 (occ-function (car occs)) netlist
                               used-modules result)))))
    (otherwise f))

  ((ord-lessp (cons (add1 (count (unbind-list used-modules netlist)))
                    (count x0)))))


(disable boundp)

(disable subnet0)


(defn subnet (flg x0 netlist)
  (subnet0 flg x0 netlist nil nil))


(disable subnet)



(defn delete-null-entries (database)
  (if (listp database)
      (let ((entry (car database))
            (rest  (cdr database)))
        (if (listp (cdr entry))
            (cons entry (delete-null-entries rest))
            (delete-null-entries rest)))
      nil))
```

```
(disable delete-null-entries)




;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;  Count of Gates, Pads, Primitives, or Transistors
;;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn primitive-count (flg x0 type netlist)
  (case flg

    (0 (let ((fn x0))

         (cond ((primp fn) (primp2 fn type))

               ((lookup-module fn netlist)
                (let ((database (netlist-properties (subnet 0 fn netlist)
                                                    (list type) T)))
                  (if (net-errorp database)
                      database
                      (value type (value fn database)))))

               (T (pred-error 'unknown-module fn)))))

    (1 (let ((body  x0)
             (props (required-props (list type))))

         (let ((database (module-database (subnet 1 body netlist) props T)))

           (value type
                  (occurrence-data props body nil nil nil database)))))

    (otherwise (pred-error 'unknown-flag (list (list 'flg flg)
                                               (list 'x0 x0))))))


(disable primitive-count)




;;; ~~~~~~~~~~~~~~~~~~~~
;;;  Output Dependencies
;;; ~~~~~~~~~~~~~~~~~~~~
```

```
(defn out-depends (flg x0 x1 x2 netlist)
  (case
    flg
    (0 (let ((fn   x0)   ; name of a primitive or a module in netlist
             (args x1))  ; list of dependency lists for fn inputs

         (cond
          ((primp fn)
           (list-union-values (primp2 fn 'out-depends)
                              (pairlist (primp2 fn 'inputs) args)))

          ((lookup-module fn netlist)
           (let ((database (netlist-properties (subnet 0 fn netlist)
                                               '(out-depends inputs) T)))
             (if (net-errorp database)
                 database
                 (let ((props (value fn database)))
                   (list-union-values (value 'out-depends props)
                                      (pairlist (value 'inputs props)
                                                args))))))

          (T (pred-error 'unknown-module fn)))))

    (1 (let ((body   x0) ; list of occurrences from a module
             (m-ins  x1)
             (m-outs x2)
             (props  (required-props '(out-depends))))

         (let ((database   (module-database (subnet 1 body netlist) props T))
               (IO-signals (intersection m-ins m-outs)))

           (let ((m-outs (mark-IO-outs m-outs IO-signals)))

             (value 'out-depends
                    (occurrence-data props body m-ins m-outs IO-signals
                                     database))))))

    (otherwise (pred-error 'unknown-flag
                           (list (list 'flg flg)
                                 (list 'x0 x0)
                                 (list 'x1 x1)
                                 (list 'x2 x2))))))


(disable out-depends)



(defn dependency-table (netlist)
  (netlist-properties netlist '(out-depends) T))
```

```
(disable dependency-table)


;;;  Function output-dependencies was never part of the predicate.


(defn output-dependencies (module-name args netlist)
  ;; ARGS is a list of dependencies for MODULE-NAME's inputs.
  (let ((x (cond ((primp module-name)
                  (length (primp2 module-name 'inputs)))
                 ((lookup-module module-name netlist)
                  (length (module-inputs (lookup-module module-name
                                                        netlist))))
                 (T (pred-error 'unknown-module module-name)))))
    (cond ((net-errorp x) x)
          ((equal (length args) x)
           (let ((r (out-depends 0 module-name args nil netlist)))
             (if (net-errorp r)
                 r
                 (list-list-sort r))))
          (T (pred-error 'wrong-number-of-args
                         (list (list module-name 'expects x)
                               (list 'got (length args) args)))))))


(disable output-dependencies)




;;; ~~~~~~~~~~~~~~~~~~~~~~~~
;;;  Input and Output Types
;;; ~~~~~~~~~~~~~~~~~~~~~~~~


(defn IO-types-collector (types alist)
  (if (listp types)
      (let ((type1 (car types)))
        (let ((real-type1 (if (listp type1)
                              (value (car type1) alist)
                              type1)))
          (cons real-type1
                (IO-types-collector (cdr types) alist))))
      nil))


(disable IO-types-collector)
```

```
(defn IO-types (flg x0 x1 x2 netlist)
  (case
    flg
    (0 (let ((fn        x0)
             (arg-types x1))

         (cond
          ((primp fn)
           (list (primp-lookup fn 'input-types)
                 (cons 'output-types
                       (IO-types-collector (primp2 fn 'output-types)
                                           (pairlist (primp2 fn 'inputs)
                                                     arg-types)))))

          ((lookup-module fn netlist)
           (let ((database (netlist-properties
                            (subnet 0 fn netlist)
                            '(input-types output-types inputs)
                            T)))
             (if (net-errorp database)
                 database

                 (let ((props (value fn database)))

                   (list (alist-entry 'input-types props)
                         (cons 'output-types
                               (IO-types-collector
                                  (value 'output-types props)
                                  (pairlist (value 'inputs props)
                                            arg-types))))))))

          (T (pred-error 'unknown-module fn)))))


    (1 (let ((body   x0)
             (m-ins  x1)
             (m-outs x2)
             (props  (required-props '(input-types output-types))))

         (let ((database   (module-database (subnet 1 body netlist) props T))
               (IO-signals (intersection m-ins m-outs)))

           (let ((m-outs (mark-IO-outs m-outs IO-signals)))

             (let ((data (occurrence-data props body m-ins m-outs IO-signals
                                          database)))

               (list (alist-entry 'input-types  data)
                     (alist-entry 'output-types data)))))))

    (otherwise (pred-error 'unknown-flag
                           (list (list 'flg flg)
                                 (list 'x0 x0)
                                 (list 'x1 x1)
```

```
                                (list 'x2 x2))))))


(disable IO-types)



(defn netlist-type-table (netlist)
  (netlist-properties netlist '(input-types output-types) T))


(disable netlist-type-table)


;; The following 2 functions are utility functions (never part of the
;; predicate).


(defn arg-types-matchp (actuals formals)
  (if (or (nlistp actuals) (nlistp formals))
      (equal actuals formals)

      (let ((actual1 (car actuals))
            (formal1 (car formals)))
        (cond ((or (unknownp actual1) (unknownp formal1))
               F)
              ((types-compatiblep actual1 formal1)
               (arg-types-matchp (cdr actuals) (cdr formals)))
              (T F)))))


(disable arg-types-matchp)


(defn arg-types-okp (fn arg-types netlist)
  (let ((IO-types (IO-types 0 fn arg-types nil netlist)))
    (if (boundp 'input-types IO-types)
        (arg-types-matchp arg-types
                          (value 'input-types IO-types))
        f)))


(disable arg-types-okp)




;;; ~~~~~~~~~~~~~
;;;  State Types
;;; ~~~~~~~~~~~~~
```

```
(defn state-type-requirement (flg x0 netlist)
  (case
    flg
    (0 (let ((fn x0))

         (cond ((primp fn)
                (if (primp-lookup fn 'state-types)
                    (primp2 fn 'state-types)
                    nil))

               ((lookup-module fn netlist)
                (let ((database (netlist-properties (subnet 0 fn netlist)
                                                    '(state-types) T)))
                  (if (net-errorp database)
                      database

                      (let ((props (value fn database)))

                        (if (boundp 'state-types props)
                            (value 'state-types props)
                            nil)))))

               (T (pred-error 'unknown-module fn)))))

    (1 (let ((body  x0)
             (props (required-props '(state-types))))

         (let ((database (module-database (subnet 1 body netlist) props T)))

           (value 'state-types
                  (occurrence-data props body nil nil nil database)))))

    (otherwise (pred-error 'unknown-flag
                           (list (list 'flg flg)
                                 (list 'x0 x0))))))


(disable state-type-requirement)



(defn netlist-state-types (netlist)
  (let ((database (netlist-properties netlist '(state-types) T)))
    (if (net-errorp database)
        database
        (delete-null-entries database))))
```

```
(disable netlist-state-types)


;; The following state-type functions are utility functions
;; (never part of the predicate).


(defn type-count (type word-type)
  (plus (count type) (count word-type)))


(prove-lemma type-count-lessp1 (rewrite)
  (implies (listp type)
           (lessp (type-count (car type) (caddr type))
                  (type-count type x))))


(prove-lemma type-count-lessp2 (rewrite)
  (implies (listp type)
           (lessp (type-count (cadr type) F)
                  (type-count type x))))


(prove-lemma type-count-lessp3 (rewrite)
  (implies (lessp (count t2) (count t1))
           (equal (lessp (type-count wtype t2)
                         (type-count t1 wtype))
                  T)))


(disable type-count)
```

```
(defn state-okp-0 (state type address-bits word-type)
  (cond ((equal type nil) (equal state nil))
        ((equal type 'boolp) (boolp state))
        ((listp type)
         (cond ((equal (car type) 'addressed-state)
                (state-okp-0 state (car type) (cadr type) (caddr type)))
               ((equal (car type) 'ram)
                (and (ramp state)
                     (state-okp-0 (ram-guts state) (cadr type) 0 F)))
               ((equal (car type) 'rom)
                (and (romp state)
                     (state-okp-0 (rom-guts state) (cadr type) 0 F)))
               ((equal (car type) 'stub)
                (and (stubp state)
                     (state-okp-0 (stub-guts state) (cadr type) 0 F)))
               ((equal (car type) 'quote)
                (state-okp-0 state (cadr type) 0 F))
               ((listp state)
                (and (state-okp-0 (car state) (car type) 0 F)
                     (state-okp-0 (cdr state) (cdr type) 0 F)))
               (T F)))
        ((equal type 'addressed-state)
         (cond ((zerop address-bits)
                (state-okp-0 state word-type 0 F))
               ((listp state)
                (and (state-okp-0 (car state) type (sub1 address-bits)
                                  word-type)
                     (state-okp-0 (cdr state) type (sub1 address-bits)
                                  word-type)))
               (T F)))
        ((equal type 'numberp) (numberp state))
        ((equal type 'number-listp) (cond ((nlistp state) (equal state nil))
                                          ((numberp (car state))
                                           (state-okp-0 (cdr state) type 0 F))
                                          (T F)))
        ((ramp type) (if (ramp state)
                         (state-okp-0 (ram-guts state) (ram-guts type) 0 F)
                         F))
        ((romp type) (if (romp state)
                         (state-okp-0 (rom-guts state) (rom-guts type) 0 F)
                         F))
        ((stubp type) (if (stubp state)
                          (state-okp-0 (stub-guts state) (stub-guts type) 0 F)
                          F))
        (T F))
  ((ord-lessp (cons (add1 (count state)) (type-count type word-type)))))


(disable state-okp-0)

(disable type-count-lessp1)

(disable type-count-lessp2)
```

```
(disable type-count-lessp3)


(defn state-okp (fn state netlist)
  (let ((type (state-type-requirement 0 fn netlist)))
    (state-okp-0 state type 0 F)))


(disable state-okp)


;;; ~~~~~~~~~~~~~~~~~~~~~
;;;  Loadings and Drives
;;; ~~~~~~~~~~~~~~~~~~~~~


(defn fix-dependent-drs (drs alist)
  (if (listp drs)
      (let ((dr1  (car drs)))
        (cons (cond ((signal-namep dr1)
                     (value dr1 alist))
                    ((and (listp dr1) (equal (car dr1) 'min))
                     (cons 'min (fix-dependent-drs (cdr dr1) alist)))
                    (T dr1))
              (fix-dependent-drs (cdr drs) alist)))
      nil))


(disable fix-dependent-drs)
```

```
(defn loadings-and-drives (flg x0 x1 x2 netlist)
  (case
    flg
    (0 (let ((fn      x0)
             (inputs  x1)
             (outputs x2))

         (cond
          ((primp fn)
           (let ((arg-map (append (pairlist (primp2 fn 'inputs) inputs)
                                  (pairlist (primp2 fn 'outputs) outputs))))
             (list (primp-lookup fn 'loadings)
                   (cons 'drives
                         (fix-dependent-drs (primp2 fn 'drives) arg-map)))))

          ((lookup-module fn netlist)
           (let ((database (netlist-properties (subnet 0 fn netlist)
                                               '(loadings drives
                                                 inputs outputs)
                                               T)))
             (if (net-errorp database)
                 database

                 (let ((props   (value fn database)))
                 (let ((arg-map (append (pairlist (value 'inputs props)
                                                  inputs)
                                        (pairlist (value 'outputs props)
                                                  outputs))))
                   (list (alist-entry 'loadings props)
                         (cons 'drives
                               (fix-dependent-drs (value 'drives props)
                                                  arg-map)))))))))

          (T (pred-error 'unknown-module fn)))))


    (1 (let ((body   x0)
             (m-ins  x1)
             (m-outs x2)
             (props  (required-props '(loadings drives))))

         (let ((database   (module-database (subnet 1 body netlist) props T))
               (IO-signals (intersection m-ins m-outs)))

           (let ((m-outs (mark-IO-outs m-outs IO-signals)))

             (let ((data (occurrence-data props body m-ins m-outs IO-signals
                                          database)))

               (list (alist-entry 'loadings data)
                     (alist-entry 'drives   data)))))))

    (otherwise (pred-error 'unknown-flag (list (list 'flg flg)
                                               (list 'x0 x0)
```

```
                                            (list 'x1 x1)
                                            (list 'x2 x2))))))
```

```
(disable loadings-and-drives)
```

```
(defn netlist-loadings-and-drives (netlist)
  (netlist-properties netlist '(loadings drives) T))
```

```
(disable netlist-loadings-and-drives)
```

14.34 "predicate.tests"

```
(defn net-with-syntax-errors () ; netlist is not proper list
  (list*
   '(m1 (in1) (out1)            ; length bad (no state); occ o1 not proper list
        ((o1 (out1) b-not (in1) . end)))
   '(m2 (in1) (out1)            ; bad annotation; bad occ length
        ((o1 (out1) b-not (in1) nil an-annotation))
        nil m2-is-the-same-as-m1)
   '(m3 (in1) (out1)            ; ok with annotation
        ((o1 (out1) b-not (in1) ((in1 . the-input) (out1 . the-output))))
        nil ((in1 . the-input) (out1 . the-output)))
   '(m3-1 (in1) (out1)          ; module and occ o1 wrong length
          ((o1 (out1) b-not (in1) (in1 . the-input) (out1 . the-output)))
          nil (in1 . the-input) (out1 . the-output))
   '(m4 (d) (d)                 ; module not proper list and name duplicated;
                                ; bad occ annotation; occ o1 has wrong
                                ; number of inputs and outputs
        ((o1 (d) m7 (d) an-annotation))
        nil . an-annotation)
   '(5 (in1) (01)               ; bad module name; bad output name
       ((o1 (01) b-not (in1)))
       nil)
   (list (index 6 1) '(in1) '(out1) ; bad module name; no occurrences
         nil
         nil)
   '(m4 (in1) (out1)            ; bad occ name; occ outputs not proper list;
                                ; occ function is unknown
        ((01 (out1 . x) not-a-module (in1)))
        nil)
   '(b-not (in1 in1 . x) (out1) ; module name is primitive; inputs not
                                ; proper list; duplicate input, and
                                ; thus signal, names;
                                ; occ o1 has wrong number of outputs
           ((o1 (out1 is1) b-nand (in1 in1)))
           nil)
   '(m7 (in1 in2 in4 te ti clk) (out1 out2 out3 out3 out4)
                                ; duplicate output name; duplicate
                                ; state name; output out4 not in
                                ; signals; in3 not in signals; state
                                ; o3 not an occurrence; duplicate occ
                                ; name; is1 duplicate signal; occ o1
                                ; inputs not proper list; first occ o2
                                ; has wrong number of inputs;
        ((o1 (out1 is1) fd1s (in1 clk te ti . x))
         (o2 (out2 is1) fd1s (in2 clk te))
         (o2 (out3 is1) fd1s (in3 clk te ti)))
        (o1 o2 o2 o3))
   '(m7-2 (a clk) (x)           ; state list of length 1
          ((o1 (x y) fd1 (a clk)))
          (o1))
   '(m8 (bus x a en pi) (bus c) ; bus is output twice
        ((o1 (bus) pullup (x))
         (o2 (bus zi po) ttl-bidirect (bus a en pi))
         (o3 (b) id (bus))
         (o4 (c) pullup (b)))
```

```
       nil)
   '(m9 (bus a en pi) (bus c)   ; occs not proper list;
                                ; bus not in signals.
       ((o1 (b c po) ttl-bidirect (bus a en pi)) . x)
       nil)
   'end))                       ; netlist not proper list

#|
```

```
(top-level-predicate (net-with-syntax-errors))
 (NET-ERROR 'NETLIST-SYNTAX-ERRORS
  (LIST
   (NET-ERROR
    '(MODULE M1)
    (LIST
     (NET-ERROR 'MODULE-FORM
                (LIST (NET-ERROR 'BAD-MODULE-LENGTH 4)))
     (NET-ERROR 'MODULE-OCCURRENCES
          (LIST (NET-ERROR '(OCCURRENCE O1)
                           (LIST (NET-ERROR 'OCCURRENCE-FORM
                                            (LIST (NET-ERROR 'NOT-PROPER-LIST
                                                             'END)))))))))
   (NET-ERROR
    '(MODULE M2)
    (LIST
     (NET-ERROR 'MODULE-OCCURRENCES
      (LIST
          (NET-ERROR '(OCCURRENCE O1)
                     (LIST (NET-ERROR 'OCCURRENCE-FORM
                                      (LIST (NET-ERROR 'BAD-OCCURRENCE-LENGTH
                                                       6)))))))
     (NET-ERROR 'MODULE-ANNOTATION
                (NET-ERROR 'NOT-ALIST
                           'M2-IS-THE-SAME-AS-M1))))
   (NET-ERROR
    '(MODULE M3-1)
    (LIST
     (NET-ERROR 'MODULE-FORM
                (LIST (NET-ERROR 'BAD-MODULE-LENGTH 7)))
     (NET-ERROR 'MODULE-OCCURRENCES
      (LIST
          (NET-ERROR '(OCCURRENCE O1)
                     (LIST (NET-ERROR 'OCCURRENCE-FORM
                                      (LIST (NET-ERROR 'BAD-OCCURRENCE-LENGTH
                                                       6)))))))))
   (NET-ERROR
    '(MODULE M4)
    (LIST
     (NET-ERROR 'MODULE-FORM
                (LIST (NET-ERROR 'NOT-PROPER-LIST
                                 'AN-ANNOTATION)))
     (NET-ERROR 'MODULE-NAME
                (LIST (NET-ERROR 'DUPLICATE-MODULE-DEFNS
                                 'M4)))
     (NET-ERROR 'MODULE-OCCURRENCES
      (LIST
       (NET-ERROR
                 '(OCCURRENCE O1)
                 (LIST (NET-ERROR 'OCCURRENCE-FUNCTION
                                  (LIST (NET-ERROR 'WRONG-NUMBER-OF-INPUTS
                                                   '(OCC-FUNCTION M7
                                                                  (EXPECTED 6)
                                                                  (GOT 1)))
```

```
                                  (NET-ERROR 'WRONG-NUMBER-OF-OUTPUTS
                                             '(OCC-FUNCTION M7
                                                            (EXPECTED 5)
                                                            (GOT 1)))))
                 (NET-ERROR 'OCCURRENCE-ANNOTATION
                            (NET-ERROR 'NOT-ALIST
                                       'AN-ANNOTATION))))))))
(NET-ERROR
 '(MODULE 5)
 (LIST
  (NET-ERROR 'MODULE-NAME
             (LIST (NET-ERROR 'BAD-NAME 5)))
  (NET-ERROR 'MODULE-OUTPUTS
             (LIST (NET-ERROR 'BAD-NAMES '(1))))
  (NET-ERROR 'MODULE-OCCURRENCES
   (LIST
     (NET-ERROR '(OCCURRENCE O1)
                (LIST (NET-ERROR 'OCCURRENCE-OUTPUTS
                                 (LIST (NET-ERROR 'BAD-NAMES '(1))))))))))
(NET-ERROR (LIST 'MODULE (INDEX 6 1))
           (LIST (NET-ERROR 'MODULE-NAME
                            (LIST (NET-ERROR 'BAD-NAME (INDEX 6 1))))
                 (NET-ERROR 'MODULE-OCCURRENCES
                            (LIST (NET-ERROR 'NO-OCCURRENCES NIL)
                                  (NET-ERROR 'OUTPUTS-NOT-IN-SIGNALS
                                             '(OUT1))))))
(NET-ERROR
 '(MODULE M4)
 (LIST
  (NET-ERROR 'MODULE-OCCURRENCES
   (LIST
      (NET-ERROR '(OCCURRENCE 1)
                 (LIST (NET-ERROR 'OCCURRENCE-NAME
                                  (NET-ERROR 'BAD-NAME 1))
                       (NET-ERROR 'OCCURRENCE-OUTPUTS
                                  (LIST (NET-ERROR 'NOT-PROPER-LIST 'X)))
                       (NET-ERROR 'UNKNOWN-OCCURRENCE-FUNCTION
                                  'NOT-A-MODULE)))))))
(NET-ERROR
 '(MODULE B-NOT)
 (LIST
  (NET-ERROR 'MODULE-NAME
             (LIST (NET-ERROR 'MODULE-NAME-SAME-AS-PRIMITIVE
                              'B-NOT)))
  (NET-ERROR 'MODULE-INPUTS
             (LIST (NET-ERROR 'DUPLICATES '(IN1))
                   (NET-ERROR 'NOT-PROPER-LIST 'X)))
  (NET-ERROR 'MODULE-OCCURRENCES
   (LIST
    (NET-ERROR
             '(OCCURRENCE O1)
             (LIST (NET-ERROR 'OCCURRENCE-FUNCTION
                              (LIST (NET-ERROR 'WRONG-NUMBER-OF-OUTPUTS
                                               '(OCC-FUNCTION B-NAND
                                                              (EXPECTED 1)
```

```
                                                              (GOT 2)))))))
      (NET-ERROR 'DUPLICATES-IN-SIGNALS
                 '(IN1))))))
  (NET-ERROR
   '(MODULE M7)
   (LIST
    (NET-ERROR 'MODULE-OUTPUTS
               (LIST (NET-ERROR 'DUPLICATES '(OUT3))))
    (NET-ERROR 'MODULE-OCCURRENCES
     (LIST
      (NET-ERROR '(OCCURRENCE O1)
                 (LIST (NET-ERROR 'OCCURRENCE-INPUTS
                                  (LIST (NET-ERROR 'NOT-PROPER-LIST 'X)))))
      (NET-ERROR
               '(OCCURRENCE O2)
               (LIST (NET-ERROR 'OCCURRENCE-FUNCTION
                                (LIST (NET-ERROR 'WRONG-NUMBER-OF-INPUTS
                                                 '(OCC-FUNCTION FD1S
                                                                (EXPECTED 4)
                                                                (GOT 3)))))))
      (NET-ERROR 'OUTPUTS-NOT-IN-SIGNALS
                 '(OUT4))
      (NET-ERROR 'NON-DEPENDS-NOT-IN-SIGNALS
                 '(IN3))
      (NET-ERROR 'STATES-NOT-IN-OCC-NAMES
                 '(O3))
      (NET-ERROR 'DUPLICATES-IN-SIGNALS
                 '(IS1))
      (NET-ERROR 'DUPLICATES-IN-OCC-NAMES
                 '(O2))))
    (NET-ERROR 'MODULE-STATENAMES
               (LIST (NET-ERROR 'DUPLICATES '(O2))))))
  (NET-ERROR '(MODULE M7-2)
             (LIST (NET-ERROR 'MODULE-STATENAMES
                              (LIST (NET-ERROR 'LIST-OF-LENGTH-1-NOT-ALLOWED
                                               '(O1))))))
  (NET-ERROR '(MODULE M8)
             (LIST (NET-ERROR 'MODULE-OCCURRENCES
                              (LIST (NET-ERROR 'DUPLICATES-IN-SIGNALS
                                               '(BUS))))))
  (NET-ERROR '(MODULE M9)
             (LIST (NET-ERROR 'MODULE-OCCURRENCES
                              (LIST (NET-ERROR 'NOT-PROPER-LIST 'X)
                                    (NET-ERROR 'IO-SIGNALS-NOT-IN-OUTPUTS
                                               '(BUS))))))
  (NET-ERROR 'NOT-PROPER-LIST 'END)))
```

```
(lsi-top-level-predicate (net-with-syntax-errors) 64 255)
 (NET-ERROR
  'LSI-NETLIST-SYNTAX-ERRORS
  (LIST
   (NET-ERROR
    '(MODULE M1)
    (LIST
     (NET-ERROR 'MODULE-FORM
                (LIST (NET-ERROR 'BAD-MODULE-LENGTH 4)))
     (NET-ERROR 'MODULE-OCCURRENCES
          (LIST (NET-ERROR '(OCCURRENCE O1)
                           (LIST (NET-ERROR 'OCCURRENCE-FORM
                                            (LIST (NET-ERROR 'NOT-PROPER-LIST
                                                             'END)))))))))
   (NET-ERROR
    '(MODULE M2)
    (LIST
     (NET-ERROR 'MODULE-OCCURRENCES
      (LIST
          (NET-ERROR '(OCCURRENCE O1)
                     (LIST (NET-ERROR 'OCCURRENCE-FORM
                                      (LIST (NET-ERROR 'BAD-OCCURRENCE-LENGTH
                                                       6)))))))
     (NET-ERROR 'MODULE-ANNOTATION
                (NET-ERROR 'NOT-ALIST
                           'M2-IS-THE-SAME-AS-M1))))
   (NET-ERROR
    '(MODULE M3-1)
    (LIST
     (NET-ERROR 'MODULE-FORM
                (LIST (NET-ERROR 'BAD-MODULE-LENGTH 7)))
     (NET-ERROR 'MODULE-OCCURRENCES
      (LIST
          (NET-ERROR '(OCCURRENCE O1)
                     (LIST (NET-ERROR 'OCCURRENCE-FORM
                                      (LIST (NET-ERROR 'BAD-OCCURRENCE-LENGTH
                                                       6)))))))))
   (NET-ERROR
    '(MODULE M4)
    (LIST
     (NET-ERROR 'MODULE-FORM
                (LIST (NET-ERROR 'NOT-PROPER-LIST
                                 'AN-ANNOTATION)))
     (NET-ERROR 'MODULE-NAME
                (LIST (NET-ERROR 'DUPLICATE-MODULE-DEFNS
                                 'M4)))
     (NET-ERROR 'MODULE-OCCURRENCES
      (LIST
       (NET-ERROR
                 '(OCCURRENCE O1)
                 (LIST (NET-ERROR 'OCCURRENCE-FUNCTION
                                  (LIST (NET-ERROR 'WRONG-NUMBER-OF-INPUTS
                                                   '(OCC-FUNCTION M7
                                                                  (EXPECTED 6)
```

```
                                                        (GOT 1)))
                                    (NET-ERROR 'WRONG-NUMBER-OF-OUTPUTS
                                               '(OCC-FUNCTION M7
                                                              (EXPECTED 5)
                                                              (GOT 1)))))
                     (NET-ERROR 'OCCURRENCE-ANNOTATION
                                (NET-ERROR 'NOT-ALIST
                                           'AN-ANNOTATION))))))))
 (NET-ERROR
  '(MODULE 5)
  (LIST
   (NET-ERROR 'MODULE-NAME
              (LIST (NET-ERROR 'LSI-NAME-NOT-LITATOM 5)))
   (NET-ERROR 'MODULE-OUTPUTS
              (LIST (NET-ERROR 'BAD-LSI-NAMES
                               (LIST (NET-ERROR 'LSI-NAME-NOT-LITATOM
                                                1)))))
   (NET-ERROR 'MODULE-OCCURRENCES
    (LIST
     (NET-ERROR
      '(OCCURRENCE O1)
      (LIST
       (NET-ERROR 'OCCURRENCE-OUTPUTS
        (LIST
         (NET-ERROR 'OCC-OUTPUTS
                    (LIST (NET-ERROR 'BAD-LSI-NAMES
                                     (LIST (NET-ERROR 'LSI-NAME-NOT-LITATOM
                                                      1)))))))))))))
 (NET-ERROR (LIST 'MODULE (INDEX 6 1))
            (LIST (NET-ERROR 'MODULE-NAME
                             (LIST (NET-ERROR 'LSI-NAME-NOT-LITATOM
                                              (INDEX 6 1))))
                  (NET-ERROR 'MODULE-OCCURRENCES
                             (LIST (NET-ERROR 'NO-OCCURRENCES NIL)
                                   (NET-ERROR 'OUTPUTS-NOT-IN-SIGNALS
                                              '(OUT1))))))
 (NET-ERROR
  '(MODULE M4)
  (LIST
   (NET-ERROR 'MODULE-OCCURRENCES
    (LIST
     (NET-ERROR
      '(OCCURRENCE 1)
      (LIST
       (NET-ERROR 'OCCURRENCE-NAME
                  (LIST (NET-ERROR 'LSI-NAME-NOT-LITATOM 1)))
       (NET-ERROR 'OCCURRENCE-OUTPUTS
                (LIST (NET-ERROR 'OCC-OUTPUTS
                                 (LIST (NET-ERROR 'NOT-PROPER-LIST 'X)))))
       (NET-ERROR 'UNKNOWN-OCCURRENCE-FUNCTION
                  'NOT-A-MODULE)))))))
 (NET-ERROR
  '(MODULE B-NOT)
  (LIST
   (NET-ERROR 'MODULE-NAME
```

```
              (LIST (NET-ERROR 'MODULE-NAME-SAME-AS-PRIMITIVE
                               'B-NOT)))
    (NET-ERROR 'MODULE-INPUTS
              (LIST (NET-ERROR 'DUPLICATES '(IN1))
                    (NET-ERROR 'NOT-PROPER-LIST 'X)))
    (NET-ERROR 'MODULE-OCCURRENCES
     (LIST
      (NET-ERROR
              '(OCCURRENCE O1)
              (LIST (NET-ERROR 'OCCURRENCE-FUNCTION
                               (LIST (NET-ERROR 'WRONG-NUMBER-OF-OUTPUTS
                                                '(OCC-FUNCTION B-NAND
                                                               (EXPECTED 1)
                                                               (GOT 2)))))))
      (NET-ERROR 'DUPLICATES-IN-SIGNALS
               '(IN1))))))
  (NET-ERROR
   '(MODULE M7)
   (LIST
    (NET-ERROR 'MODULE-OUTPUTS
              (LIST (NET-ERROR 'DUPLICATES '(OUT3))))
    (NET-ERROR 'MODULE-OCCURRENCES
     (LIST
      (NET-ERROR '(OCCURRENCE O1)
                (LIST (NET-ERROR 'OCCURRENCE-INPUTS
                                 (LIST (NET-ERROR 'NOT-PROPER-LIST 'X)))))
      (NET-ERROR
              '(OCCURRENCE O2)
              (LIST (NET-ERROR 'OCCURRENCE-FUNCTION
                               (LIST (NET-ERROR 'WRONG-NUMBER-OF-INPUTS
                                                '(OCC-FUNCTION FD1S
                                                               (EXPECTED 4)
                                                               (GOT 3)))))))
      (NET-ERROR 'OUTPUTS-NOT-IN-SIGNALS
               '(OUT4))
      (NET-ERROR 'NON-DEPENDS-NOT-IN-SIGNALS
               '(IN3))
      (NET-ERROR 'STATES-NOT-IN-OCC-NAMES
               '(O3))
      (NET-ERROR 'DUPLICATES-IN-SIGNALS
               '(IS1))
      (NET-ERROR 'DUPLICATES-IN-OCC-NAMES
               '(O2))))
    (NET-ERROR 'MODULE-STATENAMES
              (LIST (NET-ERROR 'DUPLICATES '(O2))))))
  (NET-ERROR '(MODULE M7-2)
            (LIST (NET-ERROR 'MODULE-STATENAMES
                             (LIST (NET-ERROR 'LIST-OF-LENGTH-1-NOT-ALLOWED
                                              '(O1))))))
  (NET-ERROR
   '(MODULE M8)
   (LIST
    (NET-ERROR 'MODULE-INPUTS
              (LIST (NET-ERROR 'BAD-LSI-NAMES
                               (LIST (NET-ERROR 'LSI-NAME-IS-KEYWORD
```

```
                                                 'BUS)))))
    (NET-ERROR 'MODULE-OUTPUTS
               (LIST (NET-ERROR 'BAD-LSI-NAMES
                                (LIST (NET-ERROR 'LSI-NAME-IS-KEYWORD
                                                 'BUS)))))
    (NET-ERROR 'MODULE-OCCURRENCES
     (LIST
      (NET-ERROR
       '(OCCURRENCE O1)
       (LIST
        (NET-ERROR 'OCCURRENCE-OUTPUTS
         (LIST
           (NET-ERROR 'OCC-OUTPUTS
                      (LIST (NET-ERROR 'BAD-LSI-NAMES
                                       (LIST (NET-ERROR 'LSI-NAME-IS-KEYWORD
                                                        'BUS)))))))))
      (NET-ERROR
       '(OCCURRENCE O2)
       (LIST
        (NET-ERROR 'OCCURRENCE-OUTPUTS
         (LIST
           (NET-ERROR 'OCC-OUTPUTS
                      (LIST (NET-ERROR 'BAD-LSI-NAMES
                                       (LIST (NET-ERROR 'LSI-NAME-IS-KEYWORD
                                                        'BUS)))))))
        (NET-ERROR 'OCCURRENCE-INPUTS
                   (LIST (NET-ERROR 'BAD-LSI-NAMES
                                    (LIST (NET-ERROR 'LSI-NAME-IS-KEYWORD
                                                     'BUS)))))))
      (NET-ERROR
       '(OCCURRENCE O3)
       (LIST
           (NET-ERROR 'OCCURRENCE-INPUTS
                      (LIST (NET-ERROR 'BAD-LSI-NAMES
                                       (LIST (NET-ERROR 'LSI-NAME-IS-KEYWORD
                                                        'BUS)))))))
      (NET-ERROR 'DUPLICATES-IN-SIGNALS
                 '(BUS))))))
  (NET-ERROR
   '(MODULE M9)
   (LIST
    (NET-ERROR 'MODULE-INPUTS
               (LIST (NET-ERROR 'BAD-LSI-NAMES
                                (LIST (NET-ERROR 'LSI-NAME-IS-KEYWORD
                                                 'BUS)))))
    (NET-ERROR 'MODULE-OUTPUTS
               (LIST (NET-ERROR 'BAD-LSI-NAMES
                                (LIST (NET-ERROR 'LSI-NAME-IS-KEYWORD
                                                 'BUS)))))
    (NET-ERROR 'MODULE-OCCURRENCES
     (LIST
      (NET-ERROR
       '(OCCURRENCE O1)
       (LIST
           (NET-ERROR 'OCCURRENCE-INPUTS
```

```
                     (LIST (NET-ERROR 'BAD-LSI-NAMES
                                      (LIST (NET-ERROR 'LSI-NAME-IS-KEYWORD
                                                       'BUS)))))))
       (NET-ERROR 'NOT-PROPER-LIST 'X)
       (NET-ERROR 'IO-SIGNALS-NOT-IN-OUTPUTS
                  '(BUS))))))
   (NET-ERROR 'NOT-PROPER-LIST 'END)))

|#
```

```
(defn net-with-lsi-syntax-errors ()
                              ; above should also be lsi syntax
                              ; errors
  (list
   '(m1-is-a-module-whose-name-contains-more-than-sixty-four-characters
                              ; module name too long; all input and
                              ; output names are lsi keywords; bad
                              ; occ output (~ not allowed in names)
     (hm_input clk_line pin dummy) (nc)
     ((occ1 (nc nc~) fd1s (hm_input clk_line pin dummy)))
     nil)
   (list (index 'm 2)         ; module and output names not litatoms; input
                              ; name does not begin with a letter
                              ; and contains illegal char (43)
         (list (pack '(49 65 43 . 0)) 'in2)
         (list (index 'out 1) (index 'out 2))
         (list (list 'occ1 (list (index 'out 1) (index 'out 2))
                     'm3 (list (pack '(49 65 43 . 0)) 'in2)))
         nil)
   '(m3 (in1 in2) (out1 out2)  ; occ name in1 is input
        ((in1 (out1) b-not (in1))
         (out2 (out2) b-not (in2))) ; this is ok
        nil)
   (list (pack '(77 52)) '(in1 clk te ti) '(out1 out2)
                              ; module name is not standard litatom;
                              ; occ name out1 same as one of
                              ; multiple outputs; occ name is1 is
                              ; signal name; occ o4 output o3
                              ; duplicates an occ name; fd1s is bad
                              ; occ name because it is also used as
                              ; occ function; iva cannot be a
                              ; signal name because it is the
                              ; lsi-name of an occ function (b-not)
                              ; in the module
         '((out1 (out1 is1) fd1s (in1 clk te ti))
           (is1 (is2 is3) fd1s (ti clk te in1))
           (o3 (iva is5) fd1s (is2 clk te is3))
           (fd1s (is6) b-not (is5))
           (o4 (out2 o3) fd1s (iva clk te is6)))
         nil)))


#|

(netlist-syntax-ok (net-with-lsi-syntax-errors))
 T
```

```
(lsi-top-level-predicate (net-with-lsi-syntax-errors) 64 255)
 (NET-ERROR
  'LSI-NETLIST-SYNTAX-ERRORS
  (LIST
   (NET-ERROR
   '(MODULE
          M1-IS-A-MODULE-WHOSE-NAME-CONTAINS-MORE-THAN-SIXTY-FOUR-CHARACTERS)
   (LIST
    (NET-ERROR 'MODULE-NAME
     (LIST
      (NET-ERROR
      '(BAD-LSI-NAME
          M1-IS-A-MODULE-WHOSE-NAME-CONTAINS-MORE-THAN-SIXTY-FOUR-CHARACTERS)
      (LIST (NET-ERROR 'NAME-TOO-LONG
                       '((NAME-LENGTH 66)
                         (MAX-ALLOWED-LENGTH 64)))))))
    (NET-ERROR 'MODULE-INPUTS
               (LIST (NET-ERROR 'BAD-LSI-NAMES
                                (LIST (NET-ERROR 'LSI-NAME-IS-KEYWORD
                                                 'HM_INPUT)
                                      (NET-ERROR 'LSI-NAME-IS-KEYWORD
                                                 'CLK_LINE)
                                      (NET-ERROR 'LSI-NAME-IS-KEYWORD
                                                 'PIN)
                                      (NET-ERROR 'LSI-NAME-IS-KEYWORD
                                                 'DUMMY)))))
    (NET-ERROR 'MODULE-OUTPUTS
               (LIST (NET-ERROR 'BAD-LSI-NAMES
                                (LIST (NET-ERROR 'LSI-NAME-IS-KEYWORD
                                                 'NC)))))
    (NET-ERROR 'MODULE-OCCURRENCES
     (LIST
      (NET-ERROR
       '(OCCURRENCE OCC1)
       (LIST
        (NET-ERROR 'OCCURRENCE-OUTPUTS
         (LIST
          (NET-ERROR 'OCC-OUTPUTS
           (LIST
            (NET-ERROR 'BAD-LSI-NAMES
                    (LIST (NET-ERROR 'LSI-NAME-IS-KEYWORD 'NC)
                          (NET-ERROR '(BAD-LSI-NAME NC~)
                                     (LIST (NET-ERROR 'ILLEGAL-CHARS-IN-NAME
                                                      '(126))))))))))
        (NET-ERROR 'OCCURRENCE-INPUTS
                   (LIST (NET-ERROR 'BAD-LSI-NAMES
                                    (LIST (NET-ERROR 'LSI-NAME-IS-KEYWORD
                                                     'HM_INPUT)
                                          (NET-ERROR 'LSI-NAME-IS-KEYWORD
                                                     'CLK_LINE)
                                          (NET-ERROR 'LSI-NAME-IS-KEYWORD
                                                     'PIN)
                                          (NET-ERROR 'LSI-NAME-IS-KEYWORD
                                                     'DUMMY))))))))))
```

```
(NET-ERROR
 (LIST 'MODULE (INDEX 'M 2))
 (LIST
  (NET-ERROR 'MODULE-NAME
             (LIST (NET-ERROR 'LSI-NAME-NOT-LITATOM
                              (INDEX 'M 2))))
  (NET-ERROR 'MODULE-INPUTS
   (LIST
    (NET-ERROR 'BAD-LSI-NAMES
             (LIST (NET-ERROR (LIST 'BAD-LSI-NAME
                                    (PACK '(49 65 43 . 0)))
                              (LIST (NET-ERROR 'NAME-BEGINS-WITH-NONLETTER
                                               49)
                                    (NET-ERROR 'ILLEGAL-CHARS-IN-NAME
                                               '(43))))))))
  (NET-ERROR 'MODULE-OUTPUTS
             (LIST (NET-ERROR 'BAD-LSI-NAMES
                              (LIST (NET-ERROR 'LSI-NAME-NOT-LITATOM
                                               (INDEX 'OUT 1))
                                    (NET-ERROR 'LSI-NAME-NOT-LITATOM
                                               (INDEX 'OUT 2))))))
  (NET-ERROR 'MODULE-OCCURRENCES
   (LIST
    (NET-ERROR
     '(OCCURRENCE OCC1)
     (LIST
      (NET-ERROR 'OCCURRENCE-OUTPUTS
       (LIST
        (NET-ERROR 'OCC-OUTPUTS
                   (LIST (NET-ERROR 'BAD-LSI-NAMES
                                    (LIST (NET-ERROR 'LSI-NAME-NOT-LITATOM
                                                     (INDEX 'OUT 1))
                                          (NET-ERROR 'LSI-NAME-NOT-LITATOM
                                                     (INDEX 'OUT 2))))))))
      (NET-ERROR 'OCCURRENCE-INPUTS
       (LIST
        (NET-ERROR 'BAD-LSI-NAMES
             (LIST (NET-ERROR (LIST 'BAD-LSI-NAME
                                    (PACK '(49 65 43 . 0)))
                              (LIST (NET-ERROR 'NAME-BEGINS-WITH-NONLETTER
                                               49)
                                    (NET-ERROR 'ILLEGAL-CHARS-IN-NAME
                                               '(43))))))))))))))
(NET-ERROR
 '(MODULE M3)
 (LIST
  (NET-ERROR 'MODULE-OCCURRENCES
    (LIST (NET-ERROR '(OCCURRENCE IN1)
                     (LIST (NET-ERROR 'OCCURRENCE-NAME
                                      (LIST (NET-ERROR 'OCC-NAME-IS-SIGNAL
                                                       'IN1))))))))
(NET-ERROR
 (LIST 'MODULE (PACK '(77 52)))
 (LIST
  (NET-ERROR 'MODULE-NAME
```

```
                (LIST (NET-ERROR (LIST 'BAD-LSI-NAME (PACK '(77 52)))
                                 (LIST (NET-ERROR 'NOT-STANDARD-LITATOM
                                                  '(LAST-CDR NIL))))))
     (NET-ERROR 'MODULE-OCCURRENCES
      (LIST
       (NET-ERROR
           '(OCCURRENCE OUT1)
           (LIST (NET-ERROR 'OCCURRENCE-NAME
                            (LIST (NET-ERROR 'OCC-NAME-IS-SAME-AS-OUTPUT-NAME
                                             'OUT1)))))
       (NET-ERROR '(OCCURRENCE IS1)
                  (LIST (NET-ERROR 'OCCURRENCE-NAME
                                   (LIST (NET-ERROR 'OCC-NAME-IS-SIGNAL
                                                    'IS1)))))
       (NET-ERROR '(OCCURRENCE O4)
                  (LIST (NET-ERROR 'OCCURRENCE-OUTPUTS
                                   (LIST (NET-ERROR 'OCC-OUTPUTS-IN-OCC-NAMES
                                                    '(O3))))))
       (NET-ERROR 'SIGNAL-OCC-FN-OVERLAP
                  '(IVA))
       (NET-ERROR 'OCC-NAME-OCC-FN-OVERLAP
                  '(FD1S))))))))

|#
```

```
(defn test-net1 ()
  '((m1 (clk) (q)  ; ok
        ((o1 (q qn) fd1 (q clk)))
        o1)
    (m2 (q clk) (q)
        ((o1 (q qn) fd1 (q clk)))
        o1)
    (m3 (x a en pi) (x y)
        ((o1 (x zi po) ttl-bidirect (x a en pi))
         (o2 (y) id (x)))
        nil)
    (m4 (x a en pi) (x y)
        ((o1 (y) id (x))
         (o2 (x zi po) ttl-bidirect (x a en pi)))
        nil)
    (m5 (a b c) (d)  ; ok
        ((o1 (i1) t-wire (a b))
         (o2 (d) t-wire (i1 c)))
        nil)
    (m6 (a b c) (d)  ; ok
        ((o1 (i1) id (a))
         (o2 (i2) id (b))
         (o3 (i3) t-wire (i1 i2))
         (o4 (d) t-wire (i3 c)))
        nil)
    (m7 (a b) (c d e)
        ((o1 (c) id (a))
         (o2 (d) t-wire (a b))
         (o3 (e) id (b)))
        nil)
    (m8 (a b c) (d e)
        ((o1 (d) t-wire (a b))
         (o2 (e) t-wire (b c)))
        nil)
    (m9 (a b) (c d)
        ((o1 (i1) b-not (b))
         (o2 (c) id (i1))
         (o3 (d) t-wire (i1 a)))
        nil)
    (m10 (a b) (c d)
         ((o1 (i1) b-and (a b))
          (o2 (c) t-wire (a i1))
          (o3 (d) t-wire (i1 b)))
         nil)
    (m11 (a b c) (d e)
         ((o1 (i1) b-or (a b))
          (o2 (i2) t-wire (a i1))
          (o3 (d) t-wire (i2 c))
          (o4 (e) t-wire (i1 b)))
         nil)
    (m12 (a b c d) (e f)  ; ok
         ((o1 (e) m13 (a b))
          (o2 (f) m13 (c d)))
         nil)
```

```
(m13 (a b) (c)
     ((o1 (is1) b-not (a))
      (o2 (c) t-wire (is1 b)))
     nil)
(m14 (i1 e1 i2 e2 io) (io)
     ((o1 (x) t-buf (e1 i1))
      (o2 (itemp) t-buf (e2 i2))
      (o3 (y) t-wire (itemp io))
      (o4 (z) t-wire (x y))
      (o5 (io) pullup (z)))
     nil)
(m15 (i1 e1 i2 e2 i3) (out)  ; ok
     ((o1 (z) m16 (i1 e1 i2 e2 i3))
      (o2 (out) pullup (z)))
     nil)
(m16 (i1 e1 i2 e2 io) (io)  ; ok
     ((o1 (x) t-buf (e1 i1))
      (o2 (itemp) t-buf (e2 i2))
      (o3 (y) t-wire (itemp io))
      (o4 (io) t-wire (x y)))
     nil)
(m17 (i1 e1 i2 e2 io) (io)
     ((o1 (x y) m20 (i1 e1 i2 e2 io))
      (o2 (z) t-wire (x y))
      (o3 (io) pullup (z)))
     nil)
(m18 (i1 e1 i2 e2 i3) (out)  ; ok
     ((o1 (z) m19 (i1 e1 i2 e2 i3))
      (o2 (out) pullup (z)))
     nil)
(m19 (i1 e1 i2 e2 io) (io)  ; ok
     ((o1 (x y) m20 (i1 e1 i2 e2 io))
      (o2 (io) t-wire (x y)))
     nil)
(m20 (i1 e1 i2 e2 io) (x y)  ; ok
     ((o1 (x) t-buf (e1 i1))
      (o2 (is) t-buf (e2 i2))
      (o3 (y) t-wire (is io)))
     nil)
(m21 (max min) (x)  ; ok
     ((o1 (x) t-wire (max min)))
     nil)
(m22 (a b) (min)  ; ok
     ((o1 (min) t-wire (a b)))
     nil)
(m23 (a b c d) (x)  ; ok
     ((o1 (i1) t-wire (a b))
      (o2 (i2) t-wire (c d))
      (o3 (x) t-wire (i1 i2)))
     nil)
(m24 (a b c) (d e)
     ((o1 (i1) b-not (b))
      (o2 (d) t-wire (a i1))
      (o3 (e) t-wire (i1 c)))
     nil)
```

```
(m25 (a b c) (d e f)
     ((o1 (i1) b-not (b))
      (o2 (d) t-wire (a i1))
      (o3 (e) t-wire (i1 c))
      (o4 (f) id (i1)))
     nil)
(m26 (a b e) (x)
     ((o1 (i1) t-buf (e a))
      (o2 (x) t-buf (i1 b)))
     nil)
(m27 (a b e) (x)  ; ok
     ((o1 (i1) t-buf (e a))
      (o2 (i2) pullup (i1))
      (o3 (x) t-buf (i2 b)))
     nil)
(m28 (a b c) (x y)
     ((o1 (x y) m29 (a b b c)))
     nil)
(m29 (a b c d) (x y)  ; ok
     ((o1 (x) t-wire (a b))
      (o2 (y) t-wire (c d)))
     nil)
(m30 (a b c) (d e)
     ((o1 (d) t-wire (a c))
      (o2 (is1) id (a))
      (o3 (e) t-wire (is1 b)))
     nil)
(m31 (a) (b)
     ((o1 (b) t-wire (a a)))
     nil)
(m32 (a) (b)
     ((o1 (is1) id (a))
      (o2 (is2) id (a))
      (o3 (b) t-wire (is1 is2)))
     nil)
(m33 (a b c) (x y)
     ((o1 (x y) m34 (a b b c)))
     nil)
(m34 (a b c d) (x y)  ; ok
     ((o1 (x) t-wire (a b))
      (o2 (y) t-wire (c d)))
     nil)
(m35 (x a en pi) (x y)  ; ok
     ((o1 (x zi po) ttl-bidirect (x a en pi))
      (o2 (y) pullup (x)))
     nil)
(m36 (x a en pi) (x y)  ; ok
     ((o1 (y) pullup (x))
      (o2 (x zi po) ttl-bidirect (x a en pi)))
     nil)
(m37 (x) (x)
     ((o1 (x) id (x)))
     nil)
(m38 (x en pi) (x y)
     ((o1 (i1) pullup (x))
```

```
        (o2 (i2 zi po) ttl-bidirect (x i1 en pi))
        (o3 (x) id (i2))
        (o4 (y) pullup (i2)))
       nil)
   (m39 (io x) (io)  ; ok
       ((o1 (io) t-wire (io x)))
       nil)
   (m40 (io e) (io)
       ((o1 (i1) pullup (io))
        (o2 (i2) t-buf (e i1))
        (o3 (io) t-wire (io i2)))
       nil)
   (m41 (io a) (io)
       ((o1 (i1) b-not (a))
        (o2 (io) t-wire (io i1)))
       nil)
   (m42 (bus-in a en pi) (bus-out c)  ; ok
       ((o1 (bus-temp) pullup (bus-in))
        (o2 (bus-out c) m43 (bus-temp a en pi)))
       nil)
   (m43 (bus a en pi) (bus c)
       ((o1 (bus zi po) ttl-bidirect (bus a en pi))
        (o2 (b) id (bus))
        (o3 (c) pullup (b)))
       nil)
   (m44 (bus-in a en pi) (bus-out c)  ; ok
       ((o1 (bus-temp) pullup (bus-in))
        (o2 (bus-out c) m45 (bus-temp a en pi)))
       nil)
   (m45 (bus a en pi) (bus c)  ; ok
       ((o1 (bus zi po) ttl-bidirect (bus a en pi))
        (o2 (c) pullup (bus)))
       nil)
   (m46 (a en) (x)
       ((o1 (i1) t-buf (en a))
        (o2 (x) b-not (i1)))
       nil)
   (m47 (bus a en pi) (bus c)
       ((o1 (b c po) ttl-bidirect (bus a en pi))
        (o2 (bus) pullup (b)))
       nil)
   (m48 (bus a pi) (bus)
       ((o1 (i1 i2) ttl-input (bus pi))
        (o2 (bus) t-buf (i1 a)))
       nil)
   (m49 (a en i1 i2 i3 i4 i5 i6 i7 i8 i9 i10 i11) (x)  ; ok
       ((o1 (x) m50 (a en i1 i2 i3 i4 i5 i6 i7 i8 i9 i10 i11)))
       nil)
   (m50 (a en i1 i2 i3 i4 i5 i6 i7 i8 i9 i10 i11) (x)
       ((o0 (is0) t-buf (en a)) ; drives 10
        (o1  (is1)  t-wire (is0  i1))
        (o2  (is2)  t-wire (is1  i2))
        (o3  (is3)  t-wire (is2  i3))
        (o4  (is4)  t-wire (is3  i4))
        (o5  (is5)  t-wire (is4  i5))
```

```
          (o6  (is6)  t-wire (is5  i6))
          (o7  (is7)  t-wire (is6  i7))
          (o8  (is8)  t-wire (is7  i8))
          (o9  (is9)  t-wire (is8  i9))
          (o10 (is10) t-wire (is9  i10))
          (o11 (x)    t-wire (is10 i11)))
         nil)
    ))


#|
```

```
(top-level-predicate (test-net1))
 (NET-ERROR 'NETLIST-ERRORS
  (LIST
   (NET-ERROR '(MODULE M2)
              (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                               (LIST (NET-ERROR '(IO-SIGNAL Q)
                                                '((INPUT-TYPE BOOLP)
                                                  (OUTPUT-TYPE BOOLP)))))))
   (NET-ERROR '(MODULE M3)
              (LIST (NET-ERROR 'RENAMED-IO-SIGNALS
                               (LIST (CONS 'Y (IO-OUT 'X))))))
   (NET-ERROR '(MODULE M4)
              (LIST (NET-ERROR 'RENAMED-IO-SIGNALS
                               '((Y . X)))))
   (NET-ERROR '(MODULE M7)
              (LIST (NET-ERROR 'T-WIRE-ERRORS
                               (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
                                                '(A B))))))
   (NET-ERROR '(MODULE M8)
              (LIST (NET-ERROR 'T-WIRE-ERRORS
                               (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                '(B))))))
   (NET-ERROR '(MODULE M9)
              (LIST (NET-ERROR 'T-WIRE-ERRORS
                               (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
                                                '(I1))))
                    (NET-ERROR 'IO-TYPE-CONFLICTS
                               (LIST (NET-ERROR '(SIGNAL I1)
                                                '((INPUT-TYPE TRI-STATE)
                                                  (OUTPUT-TYPE BOOLP)))))))
   (NET-ERROR '(MODULE M10)
              (LIST (NET-ERROR 'T-WIRE-ERRORS
                               (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
                                                '(B A))
                                     (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                '(I1))))
                    (NET-ERROR 'IO-TYPE-CONFLICTS
                               (LIST (NET-ERROR '(SIGNAL I1)
                                                '((INPUT-TYPE TRI-STATE)
                                                  (OUTPUT-TYPE BOOLP)))))))
   (NET-ERROR '(MODULE M11)
              (LIST (NET-ERROR 'T-WIRE-ERRORS
                               (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
                                                '(B A))
                                     (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                '(I1))))
                    (NET-ERROR 'IO-TYPE-CONFLICTS
                               (LIST (NET-ERROR '(SIGNAL I1)
                                                '((INPUT-TYPE TRI-STATE)
                                                  (OUTPUT-TYPE BOOLP)))))))
   (NET-ERROR '(MODULE M13)
              (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                               (LIST (NET-ERROR '(SIGNAL IS1)
                                                '((INPUT-TYPE TRI-STATE)
```

```
                                                (OUTPUT-TYPE BOOLP)))))))
   (NET-ERROR '(MODULE M14)
              (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                               (LIST (NET-ERROR '(IO-SIGNAL IO)
                                                '((INPUT-TYPE TRI-STATE)
                                                  (OUTPUT-TYPE BOOLP)))))))
   (NET-ERROR '(MODULE M17)
              (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                               (LIST (NET-ERROR '(IO-SIGNAL IO)
                                                '((INPUT-TYPE TRI-STATE)
                                                  (OUTPUT-TYPE BOOLP)))))))
   (NET-ERROR '(MODULE M24)
              (LIST (NET-ERROR 'T-WIRE-ERRORS
                               (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                '(I1))))
                    (NET-ERROR 'IO-TYPE-CONFLICTS
                               (LIST (NET-ERROR '(SIGNAL I1)
                                                '((INPUT-TYPE TRI-STATE)
                                                  (OUTPUT-TYPE BOOLP)))))))
   (NET-ERROR '(MODULE M25)
              (LIST (NET-ERROR 'T-WIRE-ERRORS
                               (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
                                                '(I1))
                                     (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                '(I1))))
                    (NET-ERROR 'IO-TYPE-CONFLICTS
                               (LIST (NET-ERROR '(SIGNAL I1)
                                                '((INPUT-TYPE TRI-STATE)
                                                  (OUTPUT-TYPE BOOLP)))))))
   (NET-ERROR
             '(MODULE M26)
             (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                              (LIST (NET-ERROR '(SIGNAL I1)
                                               '((INPUT-TYPE BOOLP)
                                                 (OUTPUT-TYPE TRI-STATE)))))))
   (NET-ERROR '(MODULE M28)
              (LIST (NET-ERROR 'T-WIRE-ERRORS
                               (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                '(B))))))
   (NET-ERROR '(MODULE M30)
              (LIST (NET-ERROR 'T-WIRE-ERRORS
                               (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                '(A))))))
   (NET-ERROR '(MODULE M31)
              (LIST (NET-ERROR 'T-WIRE-ERRORS
                               (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                '(A))))))
   (NET-ERROR '(MODULE M32)
              (LIST (NET-ERROR 'T-WIRE-ERRORS
                               (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                '(A))))))
   (NET-ERROR '(MODULE M33)
              (LIST (NET-ERROR 'T-WIRE-ERRORS
                               (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                '(B))))))
```

```
(NET-ERROR '(MODULE M37)
           (LIST (NET-ERROR 'RENAMED-IO-SIGNALS
                            (LIST (CONS (IO-OUT 'X) 'X)))
                 (NET-ERROR 'IO-TYPE-CONFLICTS
                            (LIST (NET-ERROR '(IO-SIGNAL X)
                                             '((INPUT-TYPE FREE)
                                               (OUTPUT-TYPE FREE)))))))
(NET-ERROR '(MODULE M38)
           (LIST (NET-ERROR 'RENAMED-IO-SIGNALS
                            (LIST (CONS (IO-OUT 'X) 'I2)))))
(NET-ERROR '(MODULE M40)
           (LIST (NET-ERROR 'T-WIRE-ERRORS
                            (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
                                             '(IO))))))
(NET-ERROR '(MODULE M41)
           (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                            (LIST (NET-ERROR '(SIGNAL I1)
                                             '((INPUT-TYPE TRI-STATE)
                                               (OUTPUT-TYPE BOOLP)))))))
(NET-ERROR '(MODULE M43)
           (LIST (NET-ERROR 'RENAMED-IO-SIGNALS
                            (LIST (CONS 'B (IO-OUT 'BUS))))))
(NET-ERROR
          '(MODULE M46)
          (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                           (LIST (NET-ERROR '(SIGNAL I1)
                                            '((INPUT-TYPE BOOLP)
                                              (OUTPUT-TYPE TRI-STATE)))))))
(NET-ERROR '(MODULE M47)
           (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                            (LIST (NET-ERROR '(IO-SIGNAL BUS)
                                             '((INPUT-TYPE TTL-TRI-STATE)
                                               (OUTPUT-TYPE BOOLP)))))))
(NET-ERROR
          '(MODULE M48)
          (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                           (LIST (NET-ERROR '(IO-SIGNAL BUS)
                                            '((INPUT-TYPE TTL)
                                              (OUTPUT-TYPE TRI-STATE)))))))
(NET-ERROR
        '(MODULE M50)
        (LIST (NET-ERROR 'BAD-DRIVES
                         (LIST (NET-ERROR '(SIGNAL ISO)
                                          '((DRIVE 10) (LOADING 11))))))))
```

```
(netlist-database (test-net1) F)
 (LIST
  '(M1 (DELAYS ((147 1171) (55 1343)))
       (DRIVES 9)
       (INPUT-TYPES CLK)
       (INPUTS CLK)
       (LOADINGS 1)
       (OUT-DEPENDS NIL)
       (OUTPUT-TYPES BOOLP)
       (OUTPUTS Q)
       (STATE-TYPES . BOOLP)
       (STATES . 01)
       (GATES . 7)
       (PADS . 0)
       (PRIMITIVES . 1)
       (TRANSISTORS . 26))
  (LIST 'M2
      (NET-ERROR '(MODULE M2)
                 (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                                  (LIST (NET-ERROR '(IO-SIGNAL Q)
                                                   '((INPUT-TYPE BOOLP)
                                                     (OUTPUT-TYPE BOOLP)))))))
      '(DELAYS ((147 1024) (55 1288)))
      '(DRIVES 10)
      (LIST 'INPUT-TYPES (UNKNOWN) 'CLK)
      '(INPUTS Q CLK)
      '(LOADINGS 1 1)
      '(OUT-DEPENDS NIL)
      (LIST 'OUTPUT-TYPES (UNKNOWN))
      '(OUTPUTS Q)
      '(STATE-TYPES . BOOLP)
      '(STATES . 01)
      '(GATES . 7)
      '(PADS . 0)
      '(PRIMITIVES . 1)
      '(TRANSISTORS . 26))
  (LIST 'M3
        (NET-ERROR '(MODULE M3)
                   (LIST (NET-ERROR 'RENAMED-IO-SIGNALS
                                    (LIST (CONS 'Y (IO-OUT 'X))))))
        (LIST 'DELAYS
              '(((61 PS-PF) 1633)
                ((41 PS-PF) 2253)
                EN A)
              (IO-OUT 'X))
        (LIST 'DRIVES '(8 MA) (IO-OUT 'X))
        '(INPUT-TYPES TTL-TRI-STATE BOOLP BOOLP PARAMETRIC)
        '(INPUTS X A EN PI)
        '(LOADINGS (3 PF) 1 1 1)
        '(OUT-DEPENDS (EN A) (EN A))
        '(OUTPUT-TYPES TTL-TRI-STATE TTL-TRI-STATE)
        '(OUTPUTS X Y)
        '(GATES . 0)
        '(PADS . 1)
```

```
      '(PRIMITIVES . 1)
      '(TRANSISTORS . 0))
 (LIST 'M4
      (NET-ERROR '(MODULE M4)
                 (LIST (NET-ERROR 'RENAMED-IO-SIGNALS
                                  '((Y . X)))))
      '(DELAYS (((61 PS-PF) 1633)
                ((41 PS-PF) 2253)
                EN A)
               X)
      '(DRIVES (8 MA) X)
      '(INPUT-TYPES TTL-TRI-STATE BOOLP BOOLP PARAMETRIC)
      '(INPUTS X A EN PI)
      '(LOADINGS (3 PF) 1 1 1)
      '(OUT-DEPENDS (EN A) (X))
      '(OUTPUT-TYPES TTL-TRI-STATE TTL-TRI-STATE)
      '(OUTPUTS X Y)
      '(GATES . 0)
      '(PADS . 1)
      '(PRIMITIVES . 1)
      '(TRANSISTORS . 0))
 '(M5 (DELAYS (OR A B C))
     (DRIVES (MIN A B C))
     (INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE)
     (INPUTS A B C)
     (LOADINGS 2 2 1)
     (OUT-DEPENDS (A B C))
     (OUTPUT-TYPES TRI-STATE)
     (OUTPUTS D)
     (GATES . 0)
     (PADS . 0)
     (PRIMITIVES . 0)
     (TRANSISTORS . 0))
 '(M6 (DELAYS (OR A B C))
     (DRIVES (MIN A B C))
     (INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE)
     (INPUTS A B C)
     (LOADINGS 2 2 1)
     (OUT-DEPENDS (A B C))
     (OUTPUT-TYPES TRI-STATE)
     (OUTPUTS D)
     (GATES . 0)
     (PADS . 0)
     (PRIMITIVES . 0)
     (TRANSISTORS . 0))
 (LIST 'M7
    (NET-ERROR '(MODULE M7)
               (LIST (NET-ERROR 'T-WIRE-ERRORS
                                (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
                                                 '(A B))))))
    '(DELAYS A (OR A B) B)
    '(DRIVES A (MIN A B) B)
    '(INPUT-TYPES TRI-STATE TRI-STATE)
    '(INPUTS A B)
    '(LOADINGS 1 1)
```

```
    '(OUT-DEPENDS (A) (A B) (B))
    '(OUTPUT-TYPES TRI-STATE TRI-STATE TRI-STATE)
    '(OUTPUTS C D E)
    '(GATES . 0)
    '(PADS . 0)
    '(PRIMITIVES . 0)
    '(TRANSISTORS . 0))
(LIST 'M8
      (NET-ERROR '(MODULE M8)
                 (LIST (NET-ERROR 'T-WIRE-ERRORS
                                  (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                   '(B))))))
      '(DELAYS (OR A B) (OR B C))
      '(DRIVES (MIN A B) (MIN B C))
      '(INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE)
      '(INPUTS A B C)
      '(LOADINGS 1 2 1)
      '(OUT-DEPENDS (A B) (B C))
      '(OUTPUT-TYPES TRI-STATE TRI-STATE)
      '(OUTPUTS D E)
      '(GATES . 0)
      '(PADS . 0)
      '(PRIMITIVES . 0)
      '(TRANSISTORS . 0))
(LIST 'M9
    (NET-ERROR '(MODULE M9)
               (LIST (NET-ERROR 'T-WIRE-ERRORS
                                (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
                                                 '(I1))))
                     (NET-ERROR 'IO-TYPE-CONFLICTS
                                (LIST (NET-ERROR '(SIGNAL I1)
                                                 '((INPUT-TYPE TRI-STATE)
                                                   (OUTPUT-TYPE BOOLP)))))))
    (LIST 'DELAYS
          '((70 305) (57 255) B)
          (LIST 'OR (UNKNOWN) 'A))
    (LIST 'DRIVES
          9
          (LIST 'MIN (UNKNOWN) 'A))
    '(INPUT-TYPES TRI-STATE BOOLP)
    '(INPUTS A B)
    '(LOADINGS 1 2)
    '(OUT-DEPENDS (B) (B A))
    (LIST 'OUTPUT-TYPES
          (UNKNOWN)
          'TRI-STATE)
    '(OUTPUTS C D)
    '(GATES . 1)
    '(PADS . 0)
    '(PRIMITIVES . 1)
    '(TRANSISTORS . 3))
(LIST 'M10
    (NET-ERROR '(MODULE M10)
               (LIST (NET-ERROR 'T-WIRE-ERRORS
                                (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
```

```
                                             '(B A))
                                  (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                             '(I1))))
                   (NET-ERROR 'IO-TYPE-CONFLICTS
                              (LIST (NET-ERROR '(SIGNAL I1)
                                             '((INPUT-TYPE TRI-STATE)
                                               (OUTPUT-TYPE BOOLP)))))))
     '(DELAYS (OR A ((144 710) (54 815) A B))
              (OR ((144 710) (54 815) A B) B))
     '(DRIVES (MIN 8 A) (MIN 8 B))
     '(INPUT-TYPES BOOLP BOOLP)
     '(INPUTS A B)
     '(LOADINGS 2 2)
     '(OUT-DEPENDS (A B) (A B))
     '(OUTPUT-TYPES TRI-STATE TRI-STATE)
     '(OUTPUTS C D)
     '(GATES . 2)
     '(PADS . 0)
     '(PRIMITIVES . 1)
     '(TRANSISTORS . 6))
 (LIST 'M11
     (NET-ERROR '(MODULE M11)
                (LIST (NET-ERROR 'T-WIRE-ERRORS
                              (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
                                             '(B A))
                                  (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                             '(I1))))
                   (NET-ERROR 'IO-TYPE-CONFLICTS
                              (LIST (NET-ERROR '(SIGNAL I1)
                                             '((INPUT-TYPE TRI-STATE)
                                               (OUTPUT-TYPE BOOLP)))))))
     '(DELAYS (OR A ((143 761) (58 993) A B) C)
              (OR ((143 761) (58 993) A B) B))
     '(DRIVES (MIN 7 A C) (MIN 7 B))
     '(INPUT-TYPES BOOLP BOOLP TRI-STATE)
     '(INPUTS A B C)
     '(LOADINGS 3 2 1)
     '(OUT-DEPENDS (A B C) (A B))
     '(OUTPUT-TYPES TRI-STATE TRI-STATE)
     '(OUTPUTS D E)
     '(GATES . 2)
     '(PADS . 0)
     '(PRIMITIVES . 1)
     '(TRANSISTORS . 6))
 '(M12 (DELAYS (OR ((70 305) (57 255) A) B)
               (OR ((70 305) (57 255) C) D))
       (DRIVES (MIN 9 B) (MIN 9 D))
       (INPUT-TYPES BOOLP TRI-STATE BOOLP TRI-STATE)
       (INPUTS A B C D)
       (LOADINGS 2 1 2 1)
       (OUT-DEPENDS (A B) (C D))
       (OUTPUT-TYPES TRI-STATE TRI-STATE)
       (OUTPUTS E F)
       (GATES . 2)
       (PADS . 0)
```

```
       (PRIMITIVES . 2)
       (TRANSISTORS . 6))
 (LIST 'M13
     (NET-ERROR '(MODULE M13)
                (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                                 (LIST (NET-ERROR '(SIGNAL IS1)
                                                  '((INPUT-TYPE TRI-STATE)
                                                    (OUTPUT-TYPE BOOLP)))))))
     '(DELAYS (OR ((70 305) (57 255) A) B))
     '(DRIVES (MIN 9 B))
     '(INPUT-TYPES BOOLP TRI-STATE)
     '(INPUTS A B)
     '(LOADINGS 2 1)
     '(OUT-DEPENDS (A B))
     '(OUTPUT-TYPES TRI-STATE)
     '(OUTPUTS C)
     '(GATES . 1)
     '(PADS . 0)
     '(PRIMITIVES . 1)
     '(TRANSISTORS . 3))
 (LIST 'M14
     (NET-ERROR '(MODULE M14)
                (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                                 (LIST (NET-ERROR '(IO-SIGNAL IO)
                                                  '((INPUT-TYPE TRI-STATE)
                                                    (OUTPUT-TYPE BOOLP)))))))
     '(DELAYS (OR ((146 605) (57 842) I1 E1)
                  ((146 751) (57 899) I2 E2)
                  IO))
     '(DRIVES (MIN 7 IO))
     (LIST 'INPUT-TYPES
           'BOOLP
           'BOOLP
           'BOOLP
           'BOOLP
           (UNKNOWN))
     '(INPUTS I1 E1 I2 E2 IO)
     '(LOADINGS 2 2 2 2 3)
     '(OUT-DEPENDS (I1 E1 I2 E2 IO))
     (LIST 'OUTPUT-TYPES (UNKNOWN))
     '(OUTPUTS IO)
     '(GATES . 6)
     '(PADS . 0)
     '(PRIMITIVES . 2)
     '(TRANSISTORS . 24))
 '(M15 (DELAYS (OR ((146 605) (57 842) I1 E1)
                   ((146 751) (57 899) I2 E2)
                   I3))
       (DRIVES (MIN 7 I3))
       (INPUT-TYPES BOOLP BOOLP BOOLP BOOLP TRI-STATE)
       (INPUTS I1 E1 I2 E2 I3)
       (LOADINGS 2 2 2 2 3)
       (OUT-DEPENDS (I1 E1 I2 E2 I3))
       (OUTPUT-TYPES BOOLP)
       (OUTPUTS OUT)
```

```
       (GATES . 6)
       (PADS . 0)
       (PRIMITIVES . 2)
       (TRANSISTORS . 24))
 '(M16 (DELAYS (OR ((146 459) (57 785) I1 E1)
                   ((146 605) (57 842) I2 E2)
                   IO))
       (DRIVES (MIN 8 IO))
       (INPUT-TYPES BOOLP BOOLP BOOLP BOOLP TRI-STATE)
       (INPUTS I1 E1 I2 E2 IO)
       (LOADINGS 2 2 2 2 2)
       (OUT-DEPENDS (I1 E1 I2 E2 IO))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS IO)
       (GATES . 6)
       (PADS . 0)
       (PRIMITIVES . 2)
       (TRANSISTORS . 24))
 (LIST 'M17
    (NET-ERROR '(MODULE M17)
               (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                                (LIST (NET-ERROR '(IO-SIGNAL IO)
                                                 '((INPUT-TYPE TRI-STATE)
                                                   (OUTPUT-TYPE BOOLP)))))))
    '(DELAYS (OR ((146 605) (57 842) I1 E1)
                 ((146 751) (57 899) I2 E2)
                 IO))
    '(DRIVES (MIN 7 IO))
    (LIST 'INPUT-TYPES
          'BOOLP
          'BOOLP
          'BOOLP
          'BOOLP
          (UNKNOWN))
    '(INPUTS I1 E1 I2 E2 IO)
    '(LOADINGS 2 2 2 2 3)
    '(OUT-DEPENDS (I1 E1 I2 E2 IO))
    (LIST 'OUTPUT-TYPES (UNKNOWN))
    '(OUTPUTS IO)
    '(GATES . 6)
    '(PADS . 0)
    '(PRIMITIVES . 2)
    '(TRANSISTORS . 24))
 '(M18 (DELAYS (OR ((146 605) (57 842) I1 E1)
                   ((146 751) (57 899) I2 E2)
                   I3))
       (DRIVES (MIN 7 I3))
       (INPUT-TYPES BOOLP BOOLP BOOLP BOOLP TRI-STATE)
       (INPUTS I1 E1 I2 E2 I3)
       (LOADINGS 2 2 2 2 3)
       (OUT-DEPENDS (I1 E1 I2 E2 I3))
       (OUTPUT-TYPES BOOLP)
       (OUTPUTS OUT)
       (GATES . 6)
       (PADS . 0)
```

```
       (PRIMITIVES . 2)
       (TRANSISTORS . 24))
 '(M19 (DELAYS (OR ((146 459) (57 785) I1 E1)
                   ((146 605) (57 842) I2 E2)
                   IO))
       (DRIVES (MIN 8 IO))
       (INPUT-TYPES BOOLP BOOLP BOOLP BOOLP TRI-STATE)
       (INPUTS I1 E1 I2 E2 IO)
       (LOADINGS 2 2 2 2 2)
       (OUT-DEPENDS (I1 E1 I2 E2 IO))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS IO)
       (GATES . 6)
       (PADS . 0)
       (PRIMITIVES . 2)
       (TRANSISTORS . 24))
 '(M20 (DELAYS ((146 313) (57 728) I1 E1)
               (OR ((146 459) (57 785) I2 E2) IO))
       (DRIVES 10 (MIN 9 IO))
       (INPUT-TYPES BOOLP BOOLP BOOLP BOOLP TRI-STATE)
       (INPUTS I1 E1 I2 E2 IO)
       (LOADINGS 2 2 2 2 1)
       (OUT-DEPENDS (I1 E1) (I2 E2 IO))
       (OUTPUT-TYPES TRI-STATE TRI-STATE)
       (OUTPUTS X Y)
       (GATES . 6)
       (PADS . 0)
       (PRIMITIVES . 2)
       (TRANSISTORS . 24))
 '(M21 (DELAYS (OR MAX MIN))
       (DRIVES (MIN MAX MIN))
       (INPUT-TYPES TRI-STATE TRI-STATE)
       (INPUTS MAX MIN)
       (LOADINGS 1 1)
       (OUT-DEPENDS (MAX MIN))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS X)
       (GATES . 0)
       (PADS . 0)
       (PRIMITIVES . 0)
       (TRANSISTORS . 0))
 '(M22 (DELAYS (OR A B))
       (DRIVES (MIN A B))
       (INPUT-TYPES TRI-STATE TRI-STATE)
       (INPUTS A B)
       (LOADINGS 1 1)
       (OUT-DEPENDS (A B))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS MIN)
       (GATES . 0)
       (PADS . 0)
       (PRIMITIVES . 0)
       (TRANSISTORS . 0))
 '(M23 (DELAYS (OR A B C D))
       (DRIVES (MIN A B C D))
```

```
        (INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE TRI-STATE)
        (INPUTS A B C D)
        (LOADINGS 2 2 2 2)
        (OUT-DEPENDS (A B C D))
        (OUTPUT-TYPES TRI-STATE)
        (OUTPUTS X)
        (GATES . 0)
        (PADS . 0)
        (PRIMITIVES . 0)
        (TRANSISTORS . 0))
  (LIST 'M24
      (NET-ERROR '(MODULE M24)
                 (LIST (NET-ERROR 'T-WIRE-ERRORS
                                  (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                   '(I1))))
                       (NET-ERROR 'IO-TYPE-CONFLICTS
                                  (LIST (NET-ERROR '(SIGNAL I1)
                                                   '((INPUT-TYPE TRI-STATE)
                                                     (OUTPUT-TYPE BOOLP)))))))
      '(DELAYS (OR A ((70 375) (57 312) B))
               (OR ((70 375) (57 312) B) C))
      '(DRIVES (MIN 8 A) (MIN 8 C))
      '(INPUT-TYPES TRI-STATE BOOLP TRI-STATE)
      '(INPUTS A B C)
      '(LOADINGS 1 2 1)
      '(OUT-DEPENDS (A B) (B C))
      '(OUTPUT-TYPES TRI-STATE TRI-STATE)
      '(OUTPUTS D E)
      '(GATES . 1)
      '(PADS . 0)
      '(PRIMITIVES . 1)
      '(TRANSISTORS . 3))
  (LIST 'M25
      (NET-ERROR '(MODULE M25)
                 (LIST (NET-ERROR 'T-WIRE-ERRORS
                                  (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
                                                   '(I1))
                                        (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                   '(I1))))
                       (NET-ERROR 'IO-TYPE-CONFLICTS
                                  (LIST (NET-ERROR '(SIGNAL I1)
                                                   '((INPUT-TYPE TRI-STATE)
                                                     (OUTPUT-TYPE BOOLP)))))))
      (LIST 'DELAYS
            (LIST 'OR 'A (UNKNOWN))
            (LIST 'OR (UNKNOWN) 'C)
            '((70 375) (57 312) B))
      (LIST 'DRIVES
            (LIST 'MIN (UNKNOWN) 'A)
            (LIST 'MIN (UNKNOWN) 'C)
            8)
      '(INPUT-TYPES TRI-STATE BOOLP TRI-STATE)
      '(INPUTS A B C)
      '(LOADINGS 1 2 1)
      '(OUT-DEPENDS (A B) (B C) (B))
```

```
     (LIST 'OUTPUT-TYPES
           'TRI-STATE
           'TRI-STATE
           (UNKNOWN))
     '(OUTPUTS D E F)
     '(GATES . 1)
     '(PADS . 0)
     '(PRIMITIVES . 1)
     '(TRANSISTORS . 3))
 (LIST 'M26
  (NET-ERROR
           '(MODULE M26)
           (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                            (LIST (NET-ERROR '(SIGNAL I1)
                                             '((INPUT-TYPE BOOLP)
                                               (OUTPUT-TYPE TRI-STATE)))))))
  '(DELAYS ((146 313)
           (57 728)
           B
           (RANGE 605 842 A E)))
  '(DRIVES 10)
  '(INPUT-TYPES BOOLP BOOLP BOOLP)
  '(INPUTS A B E)
  '(LOADINGS 2 2 2)
  '(OUT-DEPENDS (B A E))
  '(OUTPUT-TYPES TRI-STATE)
  '(OUTPUTS X)
  '(GATES . 6)
  '(PADS . 0)
  '(PRIMITIVES . 2)
  '(TRANSISTORS . 24))
 '(M27 (DELAYS ((146 313)
               (57 728)
               B
               (RANGE 751 899 A E)))
       (DRIVES 10)
       (INPUT-TYPES BOOLP BOOLP BOOLP)
       (INPUTS A B E)
       (LOADINGS 2 2 2)
       (OUT-DEPENDS (B A E))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS X)
       (GATES . 6)
       (PADS . 0)
       (PRIMITIVES . 2)
       (TRANSISTORS . 24))
 (LIST 'M28
       (NET-ERROR '(MODULE M28)
                  (LIST (NET-ERROR 'T-WIRE-ERRORS
                                   (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                    '(B))))))
       '(DELAYS (OR A B) (OR B C))
       '(DRIVES (MIN A B) (MIN B C))
       '(INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE)
       '(INPUTS A B C)
```

```
       '(LOADINGS 1 2 1)
       '(OUT-DEPENDS (A B) (B C))
       '(OUTPUT-TYPES TRI-STATE TRI-STATE)
       '(OUTPUTS X Y)
       '(GATES . 0)
       '(PADS . 0)
       '(PRIMITIVES . 0)
       '(TRANSISTORS . 0))
 '(M29 (DELAYS (OR A B) (OR C D))
       (DRIVES (MIN A B) (MIN C D))
       (INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE TRI-STATE)
       (INPUTS A B C D)
       (LOADINGS 1 1 1 1)
       (OUT-DEPENDS (A B) (C D))
       (OUTPUT-TYPES TRI-STATE TRI-STATE)
       (OUTPUTS X Y)
       (GATES . 0)
       (PADS . 0)
       (PRIMITIVES . 0)
       (TRANSISTORS . 0))
 (LIST 'M30
       (NET-ERROR '(MODULE M30)
                  (LIST (NET-ERROR 'T-WIRE-ERRORS
                                   (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                    '(A))))))
       '(DELAYS (OR A C) (OR A B))
       '(DRIVES (MIN A C) (MIN A B))
       '(INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE)
       '(INPUTS A B C)
       '(LOADINGS 2 1 1)
       '(OUT-DEPENDS (A C) (A B))
       '(OUTPUT-TYPES TRI-STATE TRI-STATE)
       '(OUTPUTS D E)
       '(GATES . 0)
       '(PADS . 0)
       '(PRIMITIVES . 0)
       '(TRANSISTORS . 0))
 (LIST 'M31
       (NET-ERROR '(MODULE M31)
                  (LIST (NET-ERROR 'T-WIRE-ERRORS
                                   (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                    '(A))))))
       (LIST 'DELAYS
             (LIST 'OR (UNKNOWN) 'A))
       (LIST 'DRIVES
             (LIST 'MIN (UNKNOWN) 'A))
       '(INPUT-TYPES TRI-STATE)
       '(INPUTS A)
       '(LOADINGS 2)
       '(OUT-DEPENDS (A))
       '(OUTPUT-TYPES TRI-STATE)
       '(OUTPUTS B)
       '(GATES . 0)
       '(PADS . 0)
       '(PRIMITIVES . 0)
```

```
       '(TRANSISTORS . 0))
 (LIST 'M32
       (NET-ERROR '(MODULE M32)
                  (LIST (NET-ERROR 'T-WIRE-ERRORS
                                   (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                    '(A))))))
       (LIST 'DELAYS
             (LIST 'OR (UNKNOWN) 'A))
       (LIST 'DRIVES
             (LIST 'MIN (UNKNOWN) 'A))
       '(INPUT-TYPES TRI-STATE)
       '(INPUTS A)
       '(LOADINGS 2)
       '(OUT-DEPENDS (A))
       '(OUTPUT-TYPES TRI-STATE)
       '(OUTPUTS B)
       '(GATES . 0)
       '(PADS . 0)
       '(PRIMITIVES . 0)
       '(TRANSISTORS . 0))
 (LIST 'M33
       (NET-ERROR '(MODULE M33)
                  (LIST (NET-ERROR 'T-WIRE-ERRORS
                                   (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                    '(B))))))
       '(DELAYS (OR A B) (OR B C))
       '(DRIVES (MIN A B) (MIN B C))
       '(INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE)
       '(INPUTS A B C)
       '(LOADINGS 1 2 1)
       '(OUT-DEPENDS (A B) (B C))
       '(OUTPUT-TYPES TRI-STATE TRI-STATE)
       '(OUTPUTS X Y)
       '(GATES . 0)
       '(PADS . 0)
       '(PRIMITIVES . 0)
       '(TRANSISTORS . 0))
 '(M34 (DELAYS (OR A B) (OR C D))
       (DRIVES (MIN A B) (MIN C D))
       (INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE TRI-STATE)
       (INPUTS A B C D)
       (LOADINGS 1 1 1 1)
       (OUT-DEPENDS (A B) (C D))
       (OUTPUT-TYPES TRI-STATE TRI-STATE)
       (OUTPUTS X Y)
       (GATES . 0)
       (PADS . 0)
       (PRIMITIVES . 0)
       (TRANSISTORS . 0))
 (LIST 'M35
       (LIST 'DELAYS
             '(((61 PS-PF) 2243)
               ((41 PS-PF) 2663)
               EN A)
             (IO-OUT 'X))
```

```
      (LIST 'DRIVES '(7 MA) (IO-OUT 'X))
      '(INPUT-TYPES TTL-TRI-STATE BOOLP BOOLP PARAMETRIC)
      '(INPUTS X A EN PI)
      '(LOADINGS (3 PF) 1 1 1)
      '(OUT-DEPENDS (EN A) (EN A))
      '(OUTPUT-TYPES TTL-TRI-STATE BOOLP)
      '(OUTPUTS X Y)
      '(GATES . 0)
      '(PADS . 1)
      '(PRIMITIVES . 1)
      '(TRANSISTORS . 0))
'(M36 (DELAYS (((61 PS-PF) 1633)
               ((41 PS-PF) 2253)
               EN A)
              X)
      (DRIVES (8 MA) X)
      (INPUT-TYPES TTL-TRI-STATE BOOLP BOOLP PARAMETRIC)
      (INPUTS X A EN PI)
      (LOADINGS (13 PF) 1 1 1)
      (OUT-DEPENDS (EN A) (X))
      (OUTPUT-TYPES TTL-TRI-STATE BOOLP)
      (OUTPUTS X Y)
      (GATES . 0)
      (PADS . 1)
      (PRIMITIVES . 1)
      (TRANSISTORS . 0))
(LIST 'M37
     (NET-ERROR '(MODULE M37)
                (LIST (NET-ERROR 'RENAMED-IO-SIGNALS
                                 (LIST (CONS (IO-OUT 'X) 'X)))
                      (NET-ERROR 'IO-TYPE-CONFLICTS
                                 (LIST (NET-ERROR '(IO-SIGNAL X)
                                                  '((INPUT-TYPE FREE)
                                                    (OUTPUT-TYPE FREE)))))))
     '(DELAYS X)
     '(DRIVES X)
     (LIST 'INPUT-TYPES (UNKNOWN))
     '(INPUTS X)
     '(LOADINGS 0)
     '(OUT-DEPENDS (X))
     (LIST 'OUTPUT-TYPES (UNKNOWN))
     '(OUTPUTS X)
     '(GATES . 0)
     '(PADS . 0)
     '(PRIMITIVES . 0)
     '(TRANSISTORS . 0))
(LIST 'M38
      (NET-ERROR '(MODULE M38)
                 (LIST (NET-ERROR 'RENAMED-IO-SIGNALS
                                  (LIST (CONS (IO-OUT 'X) 'I2)))))
      (LIST 'DELAYS
            '(((61 PS-PF) 2243)
              ((41 PS-PF) 2663)
              EN X)
            (IO-OUT 'X))
```

```
      (LIST 'DRIVES '(7 MA) (IO-OUT 'X))
      '(INPUT-TYPES TTL-TRI-STATE BOOLP PARAMETRIC)
      '(INPUTS X EN PI)
      '(LOADINGS (23 PF) 1 1)
      '(OUT-DEPENDS (EN X) (EN X))
      '(OUTPUT-TYPES TTL-TRI-STATE BOOLP)
      '(OUTPUTS X Y)
      '(GATES . 0)
      '(PADS . 1)
      '(PRIMITIVES . 1)
      '(TRANSISTORS . 0))
'(M39 (DELAYS (OR IO X))
      (DRIVES (MIN IO X))
      (INPUT-TYPES TRI-STATE TRI-STATE)
      (INPUTS IO X)
      (LOADINGS 1 1)
      (OUT-DEPENDS (IO X))
      (OUTPUT-TYPES TRI-STATE)
      (OUTPUTS IO)
      (GATES . 0)
      (PADS . 0)
      (PRIMITIVES . 0)
      (TRANSISTORS . 0))
(LIST 'M40
    (NET-ERROR '(MODULE M40)
               (LIST (NET-ERROR 'T-WIRE-ERRORS
                                (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
                                                 '(IO))))))
    '(DELAYS (OR IO ((146 459) (57 785) IO E)))
    '(DRIVES (MIN 9 IO))
    '(INPUT-TYPES TRI-STATE BOOLP)
    '(INPUTS IO E)
    '(LOADINGS 4 2)
    '(OUT-DEPENDS (IO E))
    '(OUTPUT-TYPES TRI-STATE)
    '(OUTPUTS IO)
    '(GATES . 3)
    '(PADS . 0)
    '(PRIMITIVES . 1)
    '(TRANSISTORS . 12))
(LIST 'M41
    (NET-ERROR '(MODULE M41)
               (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                                (LIST (NET-ERROR '(SIGNAL I1)
                                                 '((INPUT-TYPE TRI-STATE)
                                                   (OUTPUT-TYPE BOOLP)))))))
    '(DELAYS (OR IO ((70 305) (57 255) A)))
    '(DRIVES (MIN 9 IO))
    '(INPUT-TYPES TRI-STATE BOOLP)
    '(INPUTS IO A)
    '(LOADINGS 1 2)
    '(OUT-DEPENDS (IO A))
    '(OUTPUT-TYPES TRI-STATE)
    '(OUTPUTS IO)
    '(GATES . 1)
```

```
     '(PADS . 0)
     '(PRIMITIVES . 1)
     '(TRANSISTORS . 3))
 '(M42 (DELAYS (((61 PS-PF) 2243)
                ((41 PS-PF) 2663)
                EN A)
               BUS-OUT)
       (DRIVES (7 MA) BUS-OUT)
       (INPUT-TYPES TRI-STATE BOOLP BOOLP PARAMETRIC)
       (INPUTS BUS-IN A EN PI)
       (LOADINGS 1 1 1 1)
       (OUT-DEPENDS (EN A) (EN A))
       (OUTPUT-TYPES TTL-TRI-STATE BOOLP)
       (OUTPUTS BUS-OUT C)
       (GATES . 0)
       (PADS . 1)
       (PRIMITIVES . 1)
       (TRANSISTORS . 0))
 (LIST 'M43
       (NET-ERROR '(MODULE M43)
                  (LIST (NET-ERROR 'RENAMED-IO-SIGNALS
                                   (LIST (CONS 'B (IO-OUT 'BUS))))))
       (LIST 'DELAYS
             '(((61 PS-PF) 2243)
               ((41 PS-PF) 2663)
               EN A)
             (IO-OUT 'BUS))
       (LIST 'DRIVES '(7 MA) (IO-OUT 'BUS))
       '(INPUT-TYPES TTL-TRI-STATE BOOLP BOOLP PARAMETRIC)
       '(INPUTS BUS A EN PI)
       '(LOADINGS (3 PF) 1 1 1)
       '(OUT-DEPENDS (EN A) (EN A))
       '(OUTPUT-TYPES TTL-TRI-STATE BOOLP)
       '(OUTPUTS BUS C)
       '(GATES . 0)
       '(PADS . 1)
       '(PRIMITIVES . 1)
       '(TRANSISTORS . 0))
 '(M44 (DELAYS (((61 PS-PF) 2243)
                ((41 PS-PF) 2663)
                EN A)
               BUS-OUT)
       (DRIVES (7 MA) BUS-OUT)
       (INPUT-TYPES TRI-STATE BOOLP BOOLP PARAMETRIC)
       (INPUTS BUS-IN A EN PI)
       (LOADINGS 1 1 1 1)
       (OUT-DEPENDS (EN A) (EN A))
       (OUTPUT-TYPES TTL-TRI-STATE BOOLP)
       (OUTPUTS BUS-OUT C)
       (GATES . 0)
       (PADS . 1)
       (PRIMITIVES . 1)
       (TRANSISTORS . 0))
 (LIST 'M45
       (LIST 'DELAYS
```

```
             '(((61 PS-PF) 2243)
               ((41 PS-PF) 2663)
               EN A)
             (IO-OUT 'BUS))
       (LIST 'DRIVES '(7 MA) (IO-OUT 'BUS))
       '(INPUT-TYPES TTL-TRI-STATE BOOLP BOOLP PARAMETRIC)
       '(INPUTS BUS A EN PI)
       '(LOADINGS (3 PF) 1 1 1)
       '(OUT-DEPENDS (EN A) (EN A))
       '(OUTPUT-TYPES TTL-TRI-STATE BOOLP)
       '(OUTPUTS BUS C)
       '(GATES . 0)
       '(PADS . 1)
       '(PRIMITIVES . 1)
       '(TRANSISTORS . 0))
 (LIST 'M46
  (NET-ERROR
            '(MODULE M46)
            (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                             (LIST (NET-ERROR '(SIGNAL I1)
                                              '((INPUT-TYPE BOOLP)
                                                (OUTPUT-TYPE TRI-STATE)))))))
  '(DELAYS ((70 235)
            (57 198)
            (RANGE 605 842 A EN)))
  '(DRIVES 10)
  '(INPUT-TYPES BOOLP BOOLP)
  '(INPUTS A EN)
  '(LOADINGS 2 2)
  '(OUT-DEPENDS (A EN))
  '(OUTPUT-TYPES BOOLP)
  '(OUTPUTS X)
  '(GATES . 4)
  '(PADS . 0)
  '(PRIMITIVES . 2)
  '(TRANSISTORS . 15))
 (LIST 'M47
    (NET-ERROR '(MODULE M47)
               (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                                (LIST (NET-ERROR '(IO-SIGNAL BUS)
                                                 '((INPUT-TYPE TTL-TRI-STATE)
                                                   (OUTPUT-TYPE BOOLP)))))))
    '(DELAYS (((61 PS-PF) 2243)
              ((41 PS-PF) 2663)
              EN A)
             ((43 633) (20 638) EN A BUS))
    '(DRIVES (7 MA) 10)
    (LIST 'INPUT-TYPES
          (UNKNOWN)
          'BOOLP
          'BOOLP
          'PARAMETRIC)
    '(INPUTS BUS A EN PI)
    '(LOADINGS (3 PF) 1 1 1)
    '(OUT-DEPENDS (EN A) (EN A BUS))
```

```
    (LIST 'OUTPUT-TYPES
          (UNKNOWN)
          'BOOLP)
    '(OUTPUTS BUS C)
    '(GATES . 0)
    '(PADS . 1)
    '(PRIMITIVES . 1)
    '(TRANSISTORS . 0))
 (LIST 'M48
  (NET-ERROR
           '(MODULE M48)
           (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                            (LIST (NET-ERROR '(IO-SIGNAL BUS)
                                             '((INPUT-TYPE TTL)
                                               (OUTPUT-TYPE TRI-STATE)))))))
  '(DELAYS ((146 313)
           (57 728)
           A
           (RANGE 678 719 BUS)))
  '(DRIVES 10)
  (LIST 'INPUT-TYPES
        (UNKNOWN)
        'BOOLP
        'PARAMETRIC)
  '(INPUTS BUS A PI)
  '(LOADINGS (3 PF) 2 1)
  '(OUT-DEPENDS (A BUS))
  (LIST 'OUTPUT-TYPES (UNKNOWN))
  '(OUTPUTS BUS)
  '(GATES . 3)
  '(PADS . 1)
  '(PRIMITIVES . 2)
  '(TRANSISTORS . 12))
 '(M49 (DELAYS (OR ((146 1919) (57 1355) A EN)
                 I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11))
      (DRIVES (MIN I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11))
      (INPUT-TYPES BOOLP BOOLP TRI-STATE TRI-STATE TRI-STATE TRI-STATE
                   TRI-STATE TRI-STATE TRI-STATE TRI-STATE TRI-STATE
                   TRI-STATE TRI-STATE)
      (INPUTS A EN I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11)
      (LOADINGS 2 2 11 10 9 8 7 6 5 4 3 2 1)
      (OUT-DEPENDS (A EN I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11))
      (OUTPUT-TYPES TRI-STATE)
      (OUTPUTS X)
      (GATES . 3)
      (PADS . 0)
      (PRIMITIVES . 1)
      (TRANSISTORS . 12))
 (LIST 'M50
  (NET-ERROR
           '(MODULE M50)
           (LIST (NET-ERROR 'BAD-DRIVES
                            (LIST (NET-ERROR '(SIGNAL IS0)
                                             '((DRIVE 10) (LOADING 11))))))
  '(DELAYS (OR ((146 1919) (57 1355) A EN)
```

```
              I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11))
'(DRIVES (MIN I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11))
'(INPUT-TYPES BOOLP BOOLP TRI-STATE TRI-STATE TRI-STATE TRI-STATE
              TRI-STATE TRI-STATE TRI-STATE TRI-STATE TRI-STATE TRI-STATE
              TRI-STATE)
'(INPUTS A EN I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11)
'(LOADINGS 2 2 11 10 9 8 7 6 5 4 3 2 1)
'(OUT-DEPENDS (A EN I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11))
'(OUTPUT-TYPES TRI-STATE)
'(OUTPUTS X)
'(GATES . 3)
'(PADS . 0)
'(PRIMITIVES . 1)
'(TRANSISTORS . 12)))
```

```
(netlist-database (test-net1) T)
 (LIST
  '(M1 (DELAYS ((147 1171) (55 1343)))
       (DRIVES 9)
       (INPUT-TYPES CLK)
       (INPUTS CLK)
       (LOADINGS 1)
       (OUT-DEPENDS NIL)
       (OUTPUT-TYPES BOOLP)
       (OUTPUTS Q)
       (STATE-TYPES . BOOLP)
       (STATES . 01)
       (GATES . 7)
       (PADS . 0)
       (PRIMITIVES . 1)
       (TRANSISTORS . 26))
  (LIST 'M2
      (NET-ERROR '(MODULE M2)
                 (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                                  (LIST (NET-ERROR '(IO-SIGNAL Q)
                                                   '((INPUT-TYPE BOOLP)
                                                     (OUTPUT-TYPE BOOLP)))))))
      '(DELAYS ((147 1024) (55 1288)))
      '(DRIVES 10)
      (LIST 'INPUT-TYPES (UNKNOWN) 'CLK)
      '(INPUTS Q CLK)
      '(LOADINGS 1 1)
      '(OUT-DEPENDS NIL)
      (LIST 'OUTPUT-TYPES (UNKNOWN))
      '(OUTPUTS Q)
      '(STATE-TYPES . BOOLP)
      '(STATES . 01)
      '(GATES . 7)
      '(PADS . 0)
      '(PRIMITIVES . 1)
      '(TRANSISTORS . 26))
  (LIST 'M3
        (NET-ERROR '(MODULE M3)
                   (LIST (NET-ERROR 'RENAMED-IO-SIGNALS
                                    (LIST (CONS 'Y (IO-OUT 'X))))))
        (LIST 'DELAYS
              '(((61 PS-PF) 1633)
                ((41 PS-PF) 2253)
                EN A)
              (IO-OUT 'X))
        (LIST 'DRIVES '(8 MA) (IO-OUT 'X))
        '(INPUT-TYPES TTL-TRI-STATE BOOLP BOOLP PARAMETRIC)
        '(INPUTS X A EN PI)
        '(LOADINGS (3 PF) 1 1 1)
        '(OUT-DEPENDS (EN A) (EN A))
        '(OUTPUT-TYPES TTL-TRI-STATE TTL-TRI-STATE)
        '(OUTPUTS X Y)
        '(GATES . 0)
        '(PADS . 1)
```

```
      '(PRIMITIVES . 1)
      '(TRANSISTORS . 0))
 (LIST 'M4
      (NET-ERROR '(MODULE M4)
                 (LIST (NET-ERROR 'RENAMED-IO-SIGNALS
                                  '((Y . X)))))
      '(DELAYS (((61 PS-PF) 1633)
                ((41 PS-PF) 2253)
                EN A)
               X)
      '(DRIVES (8 MA) X)
      '(INPUT-TYPES TTL-TRI-STATE BOOLP BOOLP PARAMETRIC)
      '(INPUTS X A EN PI)
      '(LOADINGS (3 PF) 1 1 1)
      '(OUT-DEPENDS (EN A) (X))
      '(OUTPUT-TYPES TTL-TRI-STATE TTL-TRI-STATE)
      '(OUTPUTS X Y)
      '(GATES . 0)
      '(PADS . 1)
      '(PRIMITIVES . 1)
      '(TRANSISTORS . 0))
 '(M5 (DELAYS (OR A B C))
     (DRIVES (MIN A B C))
     (INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE)
     (INPUTS A B C)
     (LOADINGS 2 2 1)
     (OUT-DEPENDS (A B C))
     (OUTPUT-TYPES TRI-STATE)
     (OUTPUTS D)
     (GATES . 0)
     (PADS . 0)
     (PRIMITIVES . 0)
     (TRANSISTORS . 0))
 '(M6 (DELAYS (OR A B C))
     (DRIVES (MIN A B C))
     (INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE)
     (INPUTS A B C)
     (LOADINGS 2 2 1)
     (OUT-DEPENDS (A B C))
     (OUTPUT-TYPES TRI-STATE)
     (OUTPUTS D)
     (GATES . 0)
     (PADS . 0)
     (PRIMITIVES . 0)
     (TRANSISTORS . 0))
 (LIST 'M7
    (NET-ERROR '(MODULE M7)
               (LIST (NET-ERROR 'T-WIRE-ERRORS
                                (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
                                                 '(A B))))))
    '(DELAYS A (OR A B) B)
    '(DRIVES A (MIN A B) B)
    '(INPUT-TYPES TRI-STATE TRI-STATE)
    '(INPUTS A B)
    '(LOADINGS 1 1)
```

```
     '(OUT-DEPENDS (A) (A B) (B))
     '(OUTPUT-TYPES TRI-STATE TRI-STATE TRI-STATE)
     '(OUTPUTS C D E)
     '(GATES . 0)
     '(PADS . 0)
     '(PRIMITIVES . 0)
     '(TRANSISTORS . 0))
 (LIST 'M8
       (NET-ERROR '(MODULE M8)
                  (LIST (NET-ERROR 'T-WIRE-ERRORS
                                   (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                    '(B))))))
       '(DELAYS (OR A B) (OR B C))
       '(DRIVES (MIN A B) (MIN B C))
       '(INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE)
       '(INPUTS A B C)
       '(LOADINGS 1 2 1)
       '(OUT-DEPENDS (A B) (B C))
       '(OUTPUT-TYPES TRI-STATE TRI-STATE)
       '(OUTPUTS D E)
       '(GATES . 0)
       '(PADS . 0)
       '(PRIMITIVES . 0)
       '(TRANSISTORS . 0))
 (LIST 'M9
     (NET-ERROR '(MODULE M9)
                (LIST (NET-ERROR 'T-WIRE-ERRORS
                                 (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
                                                  '(I1))))
                      (NET-ERROR 'IO-TYPE-CONFLICTS
                                 (LIST (NET-ERROR '(SIGNAL I1)
                                                  '((INPUT-TYPE TRI-STATE)
                                                    (OUTPUT-TYPE BOOLP)))))))
     (LIST 'DELAYS
           '((70 305) (57 255) B)
           (LIST 'OR (UNKNOWN) 'A))
     (LIST 'DRIVES
           9
           (LIST 'MIN (UNKNOWN) 'A))
     '(INPUT-TYPES TRI-STATE BOOLP)
     '(INPUTS A B)
     '(LOADINGS 1 2)
     '(OUT-DEPENDS (B) (B A))
     (LIST 'OUTPUT-TYPES
           (UNKNOWN)
           'TRI-STATE)
     '(OUTPUTS C D)
     '(GATES . 1)
     '(PADS . 0)
     '(PRIMITIVES . 1)
     '(TRANSISTORS . 3))
 (LIST 'M10
     (NET-ERROR '(MODULE M10)
                (LIST (NET-ERROR 'T-WIRE-ERRORS
                                 (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
```

```
                                                   '(B A))
                                         (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                   '(I1))))
                      (NET-ERROR 'IO-TYPE-CONFLICTS
                                 (LIST (NET-ERROR '(SIGNAL I1)
                                                   '((INPUT-TYPE TRI-STATE)
                                                     (OUTPUT-TYPE BOOLP)))))))
     '(DELAYS (OR A ((144 710) (54 815) A B))
              (OR ((144 710) (54 815) A B) B))
     '(DRIVES (MIN 8 A) (MIN 8 B))
     '(INPUT-TYPES BOOLP BOOLP)
     '(INPUTS A B)
     '(LOADINGS 2 2)
     '(OUT-DEPENDS (A B) (A B))
     '(OUTPUT-TYPES TRI-STATE TRI-STATE)
     '(OUTPUTS C D)
     '(GATES . 2)
     '(PADS . 0)
     '(PRIMITIVES . 1)
     '(TRANSISTORS . 6))
 (LIST 'M11
     (NET-ERROR '(MODULE M11)
                (LIST (NET-ERROR 'T-WIRE-ERRORS
                                 (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
                                                   '(B A))
                                         (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                   '(I1))))
                      (NET-ERROR 'IO-TYPE-CONFLICTS
                                 (LIST (NET-ERROR '(SIGNAL I1)
                                                   '((INPUT-TYPE TRI-STATE)
                                                     (OUTPUT-TYPE BOOLP)))))))
     '(DELAYS (OR A ((143 761) (58 993) A B) C)
              (OR ((143 761) (58 993) A B) B))
     '(DRIVES (MIN 7 A C) (MIN 7 B))
     '(INPUT-TYPES BOOLP BOOLP TRI-STATE)
     '(INPUTS A B C)
     '(LOADINGS 3 2 1)
     '(OUT-DEPENDS (A B C) (A B))
     '(OUTPUT-TYPES TRI-STATE TRI-STATE)
     '(OUTPUTS D E)
     '(GATES . 2)
     '(PADS . 0)
     '(PRIMITIVES . 1)
     '(TRANSISTORS . 6))
 '(M12 (DELAYS (OR ((70 305) (57 255) A) B)
               (OR ((70 305) (57 255) C) D))
       (DRIVES (MIN 9 B) (MIN 9 D))
       (INPUT-TYPES BOOLP TRI-STATE BOOLP TRI-STATE)
       (INPUTS A B C D)
       (LOADINGS 2 1 2 1)
       (OUT-DEPENDS (A B) (C D))
       (OUTPUT-TYPES TRI-STATE TRI-STATE)
       (OUTPUTS E F)
       (GATES . 2)
       (PADS . 0)
```

```
       (PRIMITIVES . 2)
       (TRANSISTORS . 6))
  (LIST 'M13
      (NET-ERROR '(MODULE M13)
                 (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                                  (LIST (NET-ERROR '(SIGNAL IS1)
                                                   '((INPUT-TYPE TRI-STATE)
                                                     (OUTPUT-TYPE BOOLP)))))))
      '(DELAYS (OR ((70 305) (57 255) A) B))
      '(DRIVES (MIN 9 B))
      '(INPUT-TYPES BOOLP TRI-STATE)
      '(INPUTS A B)
      '(LOADINGS 2 1)
      '(OUT-DEPENDS (A B))
      '(OUTPUT-TYPES TRI-STATE)
      '(OUTPUTS C)
      '(GATES . 1)
      '(PADS . 0)
      '(PRIMITIVES . 1)
      '(TRANSISTORS . 3))
  (LIST 'M14
      (NET-ERROR '(MODULE M14)
                 (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                                  (LIST (NET-ERROR '(IO-SIGNAL IO)
                                                   '((INPUT-TYPE TRI-STATE)
                                                     (OUTPUT-TYPE BOOLP)))))))
      '(DELAYS (OR ((146 605) (57 842) I1 E1)
                   ((146 751) (57 899) I2 E2)
                   IO))
      '(DRIVES (MIN 7 IO))
      (LIST 'INPUT-TYPES
            'BOOLP
            'BOOLP
            'BOOLP
            'BOOLP
            (UNKNOWN))
      '(INPUTS I1 E1 I2 E2 IO)
      '(LOADINGS 2 2 2 2 3)
      '(OUT-DEPENDS (I1 E1 I2 E2 IO))
      (LIST 'OUTPUT-TYPES (UNKNOWN))
      '(OUTPUTS IO)
      '(GATES . 6)
      '(PADS . 0)
      '(PRIMITIVES . 2)
      '(TRANSISTORS . 24))
  '(M15 (DELAYS (OR ((146 605) (57 842) I1 E1)
                    ((146 751) (57 899) I2 E2)
                    I3))
        (DRIVES (MIN 7 I3))
        (INPUT-TYPES BOOLP BOOLP BOOLP BOOLP TRI-STATE)
        (INPUTS I1 E1 I2 E2 I3)
        (LOADINGS 2 2 2 2 3)
        (OUT-DEPENDS (I1 E1 I2 E2 I3))
        (OUTPUT-TYPES BOOLP)
        (OUTPUTS OUT)
```

```
      (GATES . 6)
      (PADS . 0)
      (PRIMITIVES . 2)
      (TRANSISTORS . 24))
'(M16 (DELAYS (OR ((146 459) (57 785) I1 E1)
                  ((146 605) (57 842) I2 E2)
                  IO))
      (DRIVES (MIN 8 IO))
      (INPUT-TYPES BOOLP BOOLP BOOLP BOOLP TRI-STATE)
      (INPUTS I1 E1 I2 E2 IO)
      (LOADINGS 2 2 2 2 2)
      (OUT-DEPENDS (I1 E1 I2 E2 IO))
      (OUTPUT-TYPES TRI-STATE)
      (OUTPUTS IO)
      (GATES . 6)
      (PADS . 0)
      (PRIMITIVES . 2)
      (TRANSISTORS . 24))
(LIST 'M17
    (NET-ERROR '(MODULE M17)
               (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                                (LIST (NET-ERROR '(IO-SIGNAL IO)
                                                 '((INPUT-TYPE TRI-STATE)
                                                   (OUTPUT-TYPE BOOLP)))))))
    '(DELAYS (OR ((146 605) (57 842) I1 E1)
                 ((146 751) (57 899) I2 E2)
                 IO))
    '(DRIVES (MIN 7 IO))
    (LIST 'INPUT-TYPES
          'BOOLP
          'BOOLP
          'BOOLP
          'BOOLP
          (UNKNOWN))
    '(INPUTS I1 E1 I2 E2 IO)
    '(LOADINGS 2 2 2 2 3)
    '(OUT-DEPENDS (I1 E1 I2 E2 IO))
    (LIST 'OUTPUT-TYPES (UNKNOWN))
    '(OUTPUTS IO)
    '(GATES . 6)
    '(PADS . 0)
    '(PRIMITIVES . 2)
    '(TRANSISTORS . 24))
'(M18 (DELAYS (OR ((146 605) (57 842) I1 E1)
                  ((146 751) (57 899) I2 E2)
                  I3))
      (DRIVES (MIN 7 I3))
      (INPUT-TYPES BOOLP BOOLP BOOLP BOOLP TRI-STATE)
      (INPUTS I1 E1 I2 E2 I3)
      (LOADINGS 2 2 2 2 3)
      (OUT-DEPENDS (I1 E1 I2 E2 I3))
      (OUTPUT-TYPES BOOLP)
      (OUTPUTS OUT)
      (GATES . 6)
      (PADS . 0)
```

```
       (PRIMITIVES . 2)
       (TRANSISTORS . 24))
 '(M19 (DELAYS (OR ((146 459) (57 785) I1 E1)
                   ((146 605) (57 842) I2 E2)
                   IO))
       (DRIVES (MIN 8 IO))
       (INPUT-TYPES BOOLP BOOLP BOOLP BOOLP TRI-STATE)
       (INPUTS I1 E1 I2 E2 IO)
       (LOADINGS 2 2 2 2 2)
       (OUT-DEPENDS (I1 E1 I2 E2 IO))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS IO)
       (GATES . 6)
       (PADS . 0)
       (PRIMITIVES . 2)
       (TRANSISTORS . 24))
 '(M20 (DELAYS ((146 313) (57 728) I1 E1)
               (OR ((146 459) (57 785) I2 E2) IO))
       (DRIVES 10 (MIN 9 IO))
       (INPUT-TYPES BOOLP BOOLP BOOLP BOOLP TRI-STATE)
       (INPUTS I1 E1 I2 E2 IO)
       (LOADINGS 2 2 2 2 1)
       (OUT-DEPENDS (I1 E1) (I2 E2 IO))
       (OUTPUT-TYPES TRI-STATE TRI-STATE)
       (OUTPUTS X Y)
       (GATES . 6)
       (PADS . 0)
       (PRIMITIVES . 2)
       (TRANSISTORS . 24))
 '(M21 (DELAYS (OR MAX MIN))
       (DRIVES (MIN MAX MIN))
       (INPUT-TYPES TRI-STATE TRI-STATE)
       (INPUTS MAX MIN)
       (LOADINGS 1 1)
       (OUT-DEPENDS (MAX MIN))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS X)
       (GATES . 0)
       (PADS . 0)
       (PRIMITIVES . 0)
       (TRANSISTORS . 0))
 '(M22 (DELAYS (OR A B))
       (DRIVES (MIN A B))
       (INPUT-TYPES TRI-STATE TRI-STATE)
       (INPUTS A B)
       (LOADINGS 1 1)
       (OUT-DEPENDS (A B))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS MIN)
       (GATES . 0)
       (PADS . 0)
       (PRIMITIVES . 0)
       (TRANSISTORS . 0))
 '(M23 (DELAYS (OR A B C D))
       (DRIVES (MIN A B C D))
```

```
       (INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE TRI-STATE)
       (INPUTS A B C D)
       (LOADINGS 2 2 2 2)
       (OUT-DEPENDS (A B C D))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS X)
       (GATES . 0)
       (PADS . 0)
       (PRIMITIVES . 0)
       (TRANSISTORS . 0))
 (LIST 'M24
     (NET-ERROR '(MODULE M24)
                (LIST (NET-ERROR 'T-WIRE-ERRORS
                                 (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                  '(I1))))
                      (NET-ERROR 'IO-TYPE-CONFLICTS
                                 (LIST (NET-ERROR '(SIGNAL I1)
                                                  '((INPUT-TYPE TRI-STATE)
                                                    (OUTPUT-TYPE BOOLP)))))))
     '(DELAYS (OR A ((70 375) (57 312) B))
              (OR ((70 375) (57 312) B) C))
     '(DRIVES (MIN 8 A) (MIN 8 C))
     '(INPUT-TYPES TRI-STATE BOOLP TRI-STATE)
     '(INPUTS A B C)
     '(LOADINGS 1 2 1)
     '(OUT-DEPENDS (A B) (B C))
     '(OUTPUT-TYPES TRI-STATE TRI-STATE)
     '(OUTPUTS D E)
     '(GATES . 1)
     '(PADS . 0)
     '(PRIMITIVES . 1)
     '(TRANSISTORS . 3))
 (LIST 'M25
     (NET-ERROR '(MODULE M25)
                (LIST (NET-ERROR 'T-WIRE-ERRORS
                                 (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
                                                  '(I1))
                                       (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                  '(I1))))
                      (NET-ERROR 'IO-TYPE-CONFLICTS
                                 (LIST (NET-ERROR '(SIGNAL I1)
                                                  '((INPUT-TYPE TRI-STATE)
                                                    (OUTPUT-TYPE BOOLP)))))))
     (LIST 'DELAYS
           (LIST 'OR 'A (UNKNOWN))
           (LIST 'OR (UNKNOWN) 'C)
           '((70 375) (57 312) B))
     (LIST 'DRIVES
           (LIST 'MIN (UNKNOWN) 'A)
           (LIST 'MIN (UNKNOWN) 'C)
           8)
     '(INPUT-TYPES TRI-STATE BOOLP TRI-STATE)
     '(INPUTS A B C)
     '(LOADINGS 1 2 1)
     '(OUT-DEPENDS (A B) (B C) (B))
```

```
     (LIST 'OUTPUT-TYPES
           'TRI-STATE
           'TRI-STATE
           (UNKNOWN))
     '(OUTPUTS D E F)
     '(GATES . 1)
     '(PADS . 0)
     '(PRIMITIVES . 1)
     '(TRANSISTORS . 3))
 (LIST 'M26
  (NET-ERROR
            '(MODULE M26)
            (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                             (LIST (NET-ERROR '(SIGNAL I1)
                                              '((INPUT-TYPE BOOLP)
                                                (OUTPUT-TYPE TRI-STATE)))))))
  '(DELAYS ((146 (RANGE 918 1155))
            (57 (RANGE 1333 1570))
            B A E))
  '(DRIVES 10)
  '(INPUT-TYPES BOOLP BOOLP BOOLP)
  '(INPUTS A B E)
  '(LOADINGS 2 2 2)
  '(OUT-DEPENDS (B A E))
  '(OUTPUT-TYPES TRI-STATE)
  '(OUTPUTS X)
  '(GATES . 6)
  '(PADS . 0)
  '(PRIMITIVES . 2)
  '(TRANSISTORS . 24))
 '(M27 (DELAYS ((146 (RANGE 1064 1212))
                (57 (RANGE 1479 1627))
                B A E))
       (DRIVES 10)
       (INPUT-TYPES BOOLP BOOLP BOOLP)
       (INPUTS A B E)
       (LOADINGS 2 2 2)
       (OUT-DEPENDS (B A E))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS X)
       (GATES . 6)
       (PADS . 0)
       (PRIMITIVES . 2)
       (TRANSISTORS . 24))
 (LIST 'M28
       (NET-ERROR '(MODULE M28)
                  (LIST (NET-ERROR 'T-WIRE-ERRORS
                                   (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                    '(B))))))
       '(DELAYS (OR A B) (OR B C))
       '(DRIVES (MIN A B) (MIN B C))
       '(INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE)
       '(INPUTS A B C)
       '(LOADINGS 1 2 1)
       '(OUT-DEPENDS (A B) (B C))
```

```
      '(OUTPUT-TYPES TRI-STATE TRI-STATE)
      '(OUTPUTS X Y)
      '(GATES . 0)
      '(PADS . 0)
      '(PRIMITIVES . 0)
      '(TRANSISTORS . 0))
'(M29 (DELAYS (OR A B) (OR C D))
      (DRIVES (MIN A B) (MIN C D))
      (INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE TRI-STATE)
      (INPUTS A B C D)
      (LOADINGS 1 1 1 1)
      (OUT-DEPENDS (A B) (C D))
      (OUTPUT-TYPES TRI-STATE TRI-STATE)
      (OUTPUTS X Y)
      (GATES . 0)
      (PADS . 0)
      (PRIMITIVES . 0)
      (TRANSISTORS . 0))
(LIST 'M30
      (NET-ERROR '(MODULE M30)
                 (LIST (NET-ERROR 'T-WIRE-ERRORS
                                  (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                   '(A))))))
      '(DELAYS (OR A C) (OR A B))
      '(DRIVES (MIN A C) (MIN A B))
      '(INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE)
      '(INPUTS A B C)
      '(LOADINGS 2 1 1)
      '(OUT-DEPENDS (A C) (A B))
      '(OUTPUT-TYPES TRI-STATE TRI-STATE)
      '(OUTPUTS D E)
      '(GATES . 0)
      '(PADS . 0)
      '(PRIMITIVES . 0)
      '(TRANSISTORS . 0))
(LIST 'M31
      (NET-ERROR '(MODULE M31)
                 (LIST (NET-ERROR 'T-WIRE-ERRORS
                                  (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                   '(A))))))
      (LIST 'DELAYS
            (LIST 'OR (UNKNOWN) 'A))
      (LIST 'DRIVES
            (LIST 'MIN (UNKNOWN) 'A))
      '(INPUT-TYPES TRI-STATE)
      '(INPUTS A)
      '(LOADINGS 2)
      '(OUT-DEPENDS (A))
      '(OUTPUT-TYPES TRI-STATE)
      '(OUTPUTS B)
      '(GATES . 0)
      '(PADS . 0)
      '(PRIMITIVES . 0)
      '(TRANSISTORS . 0))
(LIST 'M32
```

```
      (NET-ERROR '(MODULE M32)
                 (LIST (NET-ERROR 'T-WIRE-ERRORS
                                  (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                   '(A))))))
      (LIST 'DELAYS
            (LIST 'OR (UNKNOWN) 'A))
      (LIST 'DRIVES
            (LIST 'MIN (UNKNOWN) 'A))
      '(INPUT-TYPES TRI-STATE)
      '(INPUTS A)
      '(LOADINGS 2)
      '(OUT-DEPENDS (A))
      '(OUTPUT-TYPES TRI-STATE)
      '(OUTPUTS B)
      '(GATES . 0)
      '(PADS . 0)
      '(PRIMITIVES . 0)
      '(TRANSISTORS . 0))
 (LIST 'M33
      (NET-ERROR '(MODULE M33)
                 (LIST (NET-ERROR 'T-WIRE-ERRORS
                                  (LIST (NET-ERROR 'DUPLICATE-T-WIRE-INPUTS
                                                   '(B))))))
      '(DELAYS (OR A B) (OR B C))
      '(DRIVES (MIN A B) (MIN B C))
      '(INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE)
      '(INPUTS A B C)
      '(LOADINGS 1 2 1)
      '(OUT-DEPENDS (A B) (B C))
      '(OUTPUT-TYPES TRI-STATE TRI-STATE)
      '(OUTPUTS X Y)
      '(GATES . 0)
      '(PADS . 0)
      '(PRIMITIVES . 0)
      '(TRANSISTORS . 0))
 '(M34 (DELAYS (OR A B) (OR C D))
      (DRIVES (MIN A B) (MIN C D))
      (INPUT-TYPES TRI-STATE TRI-STATE TRI-STATE TRI-STATE)
      (INPUTS A B C D)
      (LOADINGS 1 1 1 1)
      (OUT-DEPENDS (A B) (C D))
      (OUTPUT-TYPES TRI-STATE TRI-STATE)
      (OUTPUTS X Y)
      (GATES . 0)
      (PADS . 0)
      (PRIMITIVES . 0)
      (TRANSISTORS . 0))
 (LIST 'M35
      (LIST 'DELAYS
            '(((61 PS-PF) 2243)
              ((41 PS-PF) 2663)
              EN A)
            (IO-OUT 'X))
      (LIST 'DRIVES '(7 MA) (IO-OUT 'X))
      '(INPUT-TYPES TTL-TRI-STATE BOOLP BOOLP PARAMETRIC)
```

```
       '(INPUTS X A EN PI)
       '(LOADINGS (3 PF) 1 1 1)
       '(OUT-DEPENDS (EN A) (EN A))
       '(OUTPUT-TYPES TTL-TRI-STATE BOOLP)
       '(OUTPUTS X Y)
       '(GATES . 0)
       '(PADS . 1)
       '(PRIMITIVES . 1)
       '(TRANSISTORS . 0))
 '(M36 (DELAYS (((61 PS-PF) 1633)
                ((41 PS-PF) 2253)
                EN A)
               X)
       (DRIVES (8 MA) X)
       (INPUT-TYPES TTL-TRI-STATE BOOLP BOOLP PARAMETRIC)
       (INPUTS X A EN PI)
       (LOADINGS (13 PF) 1 1 1)
       (OUT-DEPENDS (EN A) (X))
       (OUTPUT-TYPES TTL-TRI-STATE BOOLP)
       (OUTPUTS X Y)
       (GATES . 0)
       (PADS . 1)
       (PRIMITIVES . 1)
       (TRANSISTORS . 0))
 (LIST 'M37
      (NET-ERROR '(MODULE M37)
                 (LIST (NET-ERROR 'RENAMED-IO-SIGNALS
                                  (LIST (CONS (IO-OUT 'X) 'X)))
                       (NET-ERROR 'IO-TYPE-CONFLICTS
                                  (LIST (NET-ERROR '(IO-SIGNAL X)
                                                   '((INPUT-TYPE FREE)
                                                     (OUTPUT-TYPE FREE)))))))
      '(DELAYS X)
      '(DRIVES X)
      (LIST 'INPUT-TYPES (UNKNOWN))
      '(INPUTS X)
      '(LOADINGS 0)
      '(OUT-DEPENDS (X))
      (LIST 'OUTPUT-TYPES (UNKNOWN))
      '(OUTPUTS X)
      '(GATES . 0)
      '(PADS . 0)
      '(PRIMITIVES . 0)
      '(TRANSISTORS . 0))
 (LIST 'M38
       (NET-ERROR '(MODULE M38)
                  (LIST (NET-ERROR 'RENAMED-IO-SIGNALS
                                   (LIST (CONS (IO-OUT 'X) 'I2)))))
       (LIST 'DELAYS
             '(((61 PS-PF) 2243)
               ((41 PS-PF) 2663)
               EN X)
             (IO-OUT 'X))
       (LIST 'DRIVES '(7 MA) (IO-OUT 'X))
       '(INPUT-TYPES TTL-TRI-STATE BOOLP PARAMETRIC)
```

```
      '(INPUTS X EN PI)
      '(LOADINGS (23 PF) 1 1)
      '(OUT-DEPENDS (EN X) (EN X))
      '(OUTPUT-TYPES TTL-TRI-STATE BOOLP)
      '(OUTPUTS X Y)
      '(GATES . 0)
      '(PADS . 1)
      '(PRIMITIVES . 1)
      '(TRANSISTORS . 0))
'(M39 (DELAYS (OR IO X))
      (DRIVES (MIN IO X))
      (INPUT-TYPES TRI-STATE TRI-STATE)
      (INPUTS IO X)
      (LOADINGS 1 1)
      (OUT-DEPENDS (IO X))
      (OUTPUT-TYPES TRI-STATE)
      (OUTPUTS IO)
      (GATES . 0)
      (PADS . 0)
      (PRIMITIVES . 0)
      (TRANSISTORS . 0))
(LIST 'M40
    (NET-ERROR '(MODULE M40)
               (LIST (NET-ERROR 'T-WIRE-ERRORS
                                (LIST (NET-ERROR 'T-WIRE-INS-USED-ELSEWHERE
                                                 '(IO))))))
    '(DELAYS (OR IO ((146 459) (57 785) IO E)))
    '(DRIVES (MIN 9 IO))
    '(INPUT-TYPES TRI-STATE BOOLP)
    '(INPUTS IO E)
    '(LOADINGS 4 2)
    '(OUT-DEPENDS (IO E))
    '(OUTPUT-TYPES TRI-STATE)
    '(OUTPUTS IO)
    '(GATES . 3)
    '(PADS . 0)
    '(PRIMITIVES . 1)
    '(TRANSISTORS . 12))
(LIST 'M41
    (NET-ERROR '(MODULE M41)
               (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                                (LIST (NET-ERROR '(SIGNAL I1)
                                                 '((INPUT-TYPE TRI-STATE)
                                                   (OUTPUT-TYPE BOOLP)))))))
    '(DELAYS (OR IO ((70 305) (57 255) A)))
    '(DRIVES (MIN 9 IO))
    '(INPUT-TYPES TRI-STATE BOOLP)
    '(INPUTS IO A)
    '(LOADINGS 1 2)
    '(OUT-DEPENDS (IO A))
    '(OUTPUT-TYPES TRI-STATE)
    '(OUTPUTS IO)
    '(GATES . 1)
    '(PADS . 0)
    '(PRIMITIVES . 1)
```

```
    '(TRANSISTORS . 3))
 '(M42 (DELAYS (((61 PS-PF) 2243)
                ((41 PS-PF) 2663)
                EN A)
               BUS-OUT)
       (DRIVES (7 MA) BUS-OUT)
       (INPUT-TYPES TRI-STATE BOOLP BOOLP PARAMETRIC)
       (INPUTS BUS-IN A EN PI)
       (LOADINGS 1 1 1 1)
       (OUT-DEPENDS (EN A) (EN A))
       (OUTPUT-TYPES TTL-TRI-STATE BOOLP)
       (OUTPUTS BUS-OUT C)
       (GATES . 0)
       (PADS . 1)
       (PRIMITIVES . 1)
       (TRANSISTORS . 0))
 (LIST 'M43
       (NET-ERROR '(MODULE M43)
                  (LIST (NET-ERROR 'RENAMED-IO-SIGNALS
                                   (LIST (CONS 'B (IO-OUT 'BUS))))))
       (LIST 'DELAYS
             '(((61 PS-PF) 2243)
               ((41 PS-PF) 2663)
               EN A)
             (IO-OUT 'BUS))
       (LIST 'DRIVES '(7 MA) (IO-OUT 'BUS))
       '(INPUT-TYPES TTL-TRI-STATE BOOLP BOOLP PARAMETRIC)
       '(INPUTS BUS A EN PI)
       '(LOADINGS (3 PF) 1 1 1)
       '(OUT-DEPENDS (EN A) (EN A))
       '(OUTPUT-TYPES TTL-TRI-STATE BOOLP)
       '(OUTPUTS BUS C)
       '(GATES . 0)
       '(PADS . 1)
       '(PRIMITIVES . 1)
       '(TRANSISTORS . 0))
 '(M44 (DELAYS (((61 PS-PF) 2243)
                ((41 PS-PF) 2663)
                EN A)
               BUS-OUT)
       (DRIVES (7 MA) BUS-OUT)
       (INPUT-TYPES TRI-STATE BOOLP BOOLP PARAMETRIC)
       (INPUTS BUS-IN A EN PI)
       (LOADINGS 1 1 1 1)
       (OUT-DEPENDS (EN A) (EN A))
       (OUTPUT-TYPES TTL-TRI-STATE BOOLP)
       (OUTPUTS BUS-OUT C)
       (GATES . 0)
       (PADS . 1)
       (PRIMITIVES . 1)
       (TRANSISTORS . 0))
 (LIST 'M45
       (LIST 'DELAYS
             '(((61 PS-PF) 2243)
               ((41 PS-PF) 2663)
```

```
              EN A)
            (IO-OUT 'BUS))
      (LIST 'DRIVES '(7 MA) (IO-OUT 'BUS))
      '(INPUT-TYPES TTL-TRI-STATE BOOLP BOOLP PARAMETRIC)
      '(INPUTS BUS A EN PI)
      '(LOADINGS (3 PF) 1 1 1)
      '(OUT-DEPENDS (EN A) (EN A))
      '(OUTPUT-TYPES TTL-TRI-STATE BOOLP)
      '(OUTPUTS BUS C)
      '(GATES . 0)
      '(PADS . 1)
      '(PRIMITIVES . 1)
      '(TRANSISTORS . 0))
(LIST 'M46
 (NET-ERROR
           '(MODULE M46)
           (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                            (LIST (NET-ERROR '(SIGNAL I1)
                                             '((INPUT-TYPE BOOLP)
                                               (OUTPUT-TYPE TRI-STATE))))))
 '(DELAYS ((70 (RANGE 840 1077))
           (57 (RANGE 803 1040))
           A EN))
 '(DRIVES 10)
 '(INPUT-TYPES BOOLP BOOLP)
 '(INPUTS A EN)
 '(LOADINGS 2 2)
 '(OUT-DEPENDS (A EN))
 '(OUTPUT-TYPES BOOLP)
 '(OUTPUTS X)
 '(GATES . 4)
 '(PADS . 0)
 '(PRIMITIVES . 2)
 '(TRANSISTORS . 15))
(LIST 'M47
   (NET-ERROR '(MODULE M47)
              (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                               (LIST (NET-ERROR '(IO-SIGNAL BUS)
                                                '((INPUT-TYPE TTL-TRI-STATE)
                                                  (OUTPUT-TYPE BOOLP)))))))
   '(DELAYS (((61 PS-PF) 2243)
             ((41 PS-PF) 2663)
             EN A)
            ((43 633) (20 638) EN A BUS))
   '(DRIVES (7 MA) 10)
   (LIST 'INPUT-TYPES
         (UNKNOWN)
         'BOOLP
         'BOOLP
         'PARAMETRIC)
   '(INPUTS BUS A EN PI)
   '(LOADINGS (3 PF) 1 1 1)
   '(OUT-DEPENDS (EN A) (EN A BUS))
   (LIST 'OUTPUT-TYPES
         (UNKNOWN)
```

```
          'BOOLP)
    '(OUTPUTS BUS C)
    '(GATES . 0)
    '(PADS . 1)
    '(PRIMITIVES . 1)
    '(TRANSISTORS . 0))
  (LIST 'M48
   (NET-ERROR
            '(MODULE M48)
            (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                             (LIST (NET-ERROR '(IO-SIGNAL BUS)
                                              '((INPUT-TYPE TTL)
                                                (OUTPUT-TYPE TRI-STATE)))))))
   '(DELAYS ((146 (RANGE 991 1032))
            (57 (RANGE 1406 1447))
            A BUS))
   '(DRIVES 10)
   (LIST 'INPUT-TYPES
         (UNKNOWN)
         'BOOLP
         'PARAMETRIC)
   '(INPUTS BUS A PI)
   '(LOADINGS (3 PF) 2 1)
   '(OUT-DEPENDS (A BUS))
   (LIST 'OUTPUT-TYPES (UNKNOWN))
   '(OUTPUTS BUS)
   '(GATES . 3)
   '(PADS . 1)
   '(PRIMITIVES . 2)
   '(TRANSISTORS . 12))
  '(M49 (DELAYS (OR ((146 1919) (57 1355) A EN)
                   I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11))
        (DRIVES (MIN I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11))
        (INPUT-TYPES BOOLP BOOLP TRI-STATE TRI-STATE TRI-STATE TRI-STATE
                     TRI-STATE TRI-STATE TRI-STATE TRI-STATE TRI-STATE
                     TRI-STATE TRI-STATE)
        (INPUTS A EN I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11)
        (LOADINGS 2 2 11 10 9 8 7 6 5 4 3 2 1)
        (OUT-DEPENDS (A EN I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11))
        (OUTPUT-TYPES TRI-STATE)
        (OUTPUTS X)
        (GATES . 3)
        (PADS . 0)
        (PRIMITIVES . 1)
        (TRANSISTORS . 12))
  (LIST 'M50
   (NET-ERROR
            '(MODULE M50)
            (LIST (NET-ERROR 'BAD-DRIVES
                             (LIST (NET-ERROR '(SIGNAL IS0)
                                              '((DRIVE 10) (LOADING 11)))))))
   '(DELAYS (OR ((146 1919) (57 1355) A EN)
               I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11))
   '(DRIVES (MIN I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11))
   '(INPUT-TYPES BOOLP BOOLP TRI-STATE TRI-STATE TRI-STATE TRI-STATE
```

```
                 TRI-STATE TRI-STATE TRI-STATE TRI-STATE TRI-STATE TRI-STATE
                 TRI-STATE)
   '(INPUTS A EN I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11)
   '(LOADINGS 2 2 11 10 9 8 7 6 5 4 3 2 1)
   '(OUT-DEPENDS (A EN I1 I2 I3 I4 I5 I6 I7 I8 I9 I10 I11))
   '(OUTPUT-TYPES TRI-STATE)
   '(OUTPUTS X)
   '(GATES . 3)
   '(PADS . 0)
   '(PRIMITIVES . 1)
   '(TRANSISTORS . 12)))

|#

#|


(top-level-predicate 0)
 (NET-ERROR 'NETLIST-SYNTAX-ERRORS
            (LIST (NET-ERROR 'NOT-PROPER-LIST 0)))
|#


#|


(defn make-ram-state (structure width bit-value)
  (if (listp structure)
      (cons (make-ram-state (car structure) width bit-value)
          (make-ram-state (cdr structure) width bit-value))
    (ram (make-list width bit-value))))


(disable make-ram-state)

|#

#|

(setq d2 '((d2 (in0)
               (out0)
               ((g (out0) b-and (in0 out0)))
               nil)))


(netlist-syntax-ok d2)
 T

(lsi-netlist-syntax-ok (lisp-netlist d2) 10 50)
 T
```

```
(out-depends 0 'd2 '((a)) nil d2)
 (NET-ERROR 'NETLIST-ERRORS
            (LIST (NET-ERROR '(MODULE D2)
                             (LIST (NET-ERROR 'UNBOUND-OUTPUT-DEPENDENCIES
                                              (LIST (NET-ERROR '(OUTPUT OUT0)
                                                               '(OUT0))))))))



(setq ex-net
      '((m1 (clk)
            (out1 out2)
            ((o1 (out1 out2) fd1 (out1 clk)))
            o1)))


(top-level-predicate ex-net)
 T

(netlist-loadings-and-drives ex-net)
 '((M1 (LOADINGS 1) (DRIVES 9 10)))


(netlist-properties ex-net '(delays) F)

(netlist-properties ex-net '(delays) T)
 '((M1 (DELAYS ((147 1171) (55 1343))
               ((145 1432) (53 1447)))))



(setq ex-net '((m3 (in0 clk) (out1 out2)
                   ((o1 (oi1) b-not (in0))
                    (o2 (out1) b-not (in0))
                    (o5 (out2 waste) fd1 (oi2 clk))
                                ; oi2 is ok here because out2 does
                                ; not depend on oi2.  Out2 depends
                                ; only on fd1's internal state
                                ; before update.
                    (o4 (ii1) b-and (out2 oi1))
                    (o3 (oi2) b-and (in0 ii1)))
                   o5)))


(netlist-syntax-ok ex-net)
 T

(lsi-netlist-syntax-ok (lisp-netlist ex-net) 10 50)
 T

(out-depends 0 'm3 '((new-in) (new-clk)) nil ex-net)
 '((NEW-IN) NIL)
```

```
(list (arg-types-okp 'm3 '(boolp clk) ex-net)    ; T
      (arg-types-okp 'm3 '(clk boolp) ex-net)    ; F
      (arg-types-okp 'm3 '(boolp boolp) ex-net)) ; F


(netlist-type-table ex-net)
 '((M3 (INPUT-TYPES BOOLP CLK)
       (OUTPUT-TYPES BOOLP BOOLP)))


(netlist-state-types ex-net)
 '((M3 (STATE-TYPES . BOOLP)))

; in following, first 2 T, remaining 3 F

(list (state-okp 'm3 f ex-net)
      (state-okp 'm3 t ex-net)
      (state-okp 'm3 (x) ex-net)
      (state-okp 'm3 (list t) ex-net)
      (state-okp 'm3 nil ex-net))


(netlist-loadings-and-drives ex-net)
 '((M3 (LOADINGS 5 1) (DRIVES 10 9)))


(netlist-properties ex-net '(delays) F)

(netlist-properties ex-net '(delays) T)
 '((M3 (DELAYS ((70 235) (57 198) IN0)
               ((147 1171) (55 1343)))))



(setq ex-net '((m3 (in0 clk) (out1 out2)
                   ((o1 (is1) id (in0))
                    (o2 (is2) id (clk))
                    (o3 (is3 is4) fd1 (is1 is2))
                                ; oi2 is ok here because out2 does
                                ; not depend on oi2.  Out2 depends
                                ; only on fd1's internal state
                                ; before update.
                    (o4 (out1) id (is3))
                    (o5 (out2) id (is4)))
                   o3)))


(list (arg-types-okp 'm3 '(boolp clk) ex-net)    ; T
      (arg-types-okp 'm3 '(clk boolp) ex-net)    ; F
      (arg-types-okp 'm3 '(boolp boolp) ex-net)) ; F
```

```
(netlist-type-table ex-net)
 '((M3 (INPUT-TYPES BOOLP CLK)
       (OUTPUT-TYPES BOOLP BOOLP)))


(netlist-loadings-and-drives ex-net)
 '((M3 (LOADINGS 1 1) (DRIVES 10 10)))


(netlist-properties ex-net '(delays) F)

(netlist-properties ex-net '(delays) T)
 '((M3 (DELAYS ((147 1024) (55 1288))
               ((145 1432) (53 1447)))))



(setq ex-net '((m1 (in0) (out0)
                   ((o1 (is1) id (in0))
                    (o2 (is2) id (is1))
                    (o3 (is3) id (is2))
                    (o4 (out0) id (is3)))
                   nil)))

; all of following T

(list (arg-types-okp 'm1 '(boolp) ex-net)
      (arg-types-okp 'm1 '(clk) ex-net)
      (arg-types-okp 'm1 '(parametric) ex-net)
      (arg-types-okp 'm1 '(ttl) ex-net))


(netlist-type-table ex-net)
 '((M1 (INPUT-TYPES FREE)
       (OUTPUT-TYPES (IN0))))


(netlist-state-types ex-net)
 NIL

; in following, first T, other 2 F

(list (state-okp 'm1 nil ex-net)
      (state-okp 'm1 t ex-net)
      (state-okp 'm1 (x) ex-net))


(netlist-loadings-and-drives ex-net)
 '((M1 (LOADINGS 0) (DRIVES IN0)))


(netlist-properties ex-net '(delays) F)
```

```
(netlist-properties ex-net '(delays) T)
 '((M1 (DELAYS IN0)))



(setq test-net '((test () (a b)
                       ((inv1 (a) b-not   (b))
                        (inv2 (b) b-not   (a)))
                       nil)))


(netlist-syntax-ok test-net)
 T

(lsi-netlist-syntax-ok (lisp-netlist test-net) 10 50)
 T

(out-depends 0 'test '() nil test-net)
 (NET-ERROR 'NETLIST-ERRORS
    (LIST (NET-ERROR '(MODULE TEST)
                     (LIST (NET-ERROR 'UNBOUND-OUTPUT-DEPENDENCIES
                                      (LIST (NET-ERROR '(OUTPUT A) '(B))))))))

;?? (netlist-state-input-okp test-net (dependency-table test-net))




(setq netlist '((m2 (clk en0 en1 sel0 sel1 d0 d1) (q)
                    ((occ0 (q0) m1     (clk en0 sel0 d0 q))
                     (occ1 (q1) m1     (clk en1 sel1 d1 q))
                     (wire (q)  t-wire (q0 q1)))
                    (occ0 occ1))
                (m1 (clk en sel d q) (q)
                    ((mux   (b)    b-if  (sel d q))
                     (latch (a an) fd1   (b clk))
                     (tbuf  (q)    t-buf (en a)))
                    latch)))


(netlist-syntax-ok netlist)
 T

(top-level-predicate netlist)
 (NET-ERROR 'NETLIST-ERRORS
  (LIST
   (NET-ERROR
           '(MODULE M1)
           (LIST (NET-ERROR 'IO-TYPE-CONFLICTS
                            (LIST (NET-ERROR '(IO-SIGNAL Q)
                                             '((INPUT-TYPE BOOLP)
                                               (OUTPUT-TYPE TRI-STATE)))))))))
```

```
(setq args '((a) (b) (c) (d) (e)))

(dependency-table netlist)
 '((M2 (OUT-DEPENDS (EN0 EN1)))
   (M1 (OUT-DEPENDS (EN))))


(out-depends 0 'm1 args nil netlist)
 '((B))



(setq netlist '((m2 (clk en0 en1 sel0 sel1 d0 d1) (q)
                    ((occ0 (q0) m1     (clk en0 sel0 d0 q))
                     (occ1 (q1) m1     (clk en1 sel1 d1 q))
                     (wire (q)  t-wire (q0 q1)))
                    (occ0 occ1))
                (m1 (clk en sel d q) (q)
                    ((qpullup (qb) pullup (q))
                     (mux   (b)    b-if  (sel d qb))
                     (latch (a an) fd1   (b clk))
                     (tbuf  (q)    t-buf (en a)))
                    latch)))


(top-level-predicate netlist)
 T
```

```
(netlist-database netlist F)
 '((M2 (DELAYS (OR ((146 1043)
                    (57 1013)
                    (RANGE 1318 1398)
                    EN0)
                   ((146 1043)
                    (57 1013)
                    (RANGE 1318 1398)
                    EN1)))
       (DRIVES 5)
       (INPUT-TYPES CLK BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP)
       (INPUTS CLK EN0 EN1 SEL0 SEL1 D0 D1)
       (LOADINGS 2 2 2 2 2 1 1)
       (OUT-DEPENDS (EN0 EN1))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS Q)
       (STATE-TYPES BOOLP BOOLP)
       (STATES OCC0 OCC1)
       (GATES . 28)
       (PADS . 0)
       (PRIMITIVES . 6)
       (TRANSISTORS . 100))
   (M1 (DELAYS ((146 313)
                (57 728)
                (RANGE 1318 1398)
                EN))
       (DRIVES 10)
       (INPUT-TYPES CLK BOOLP BOOLP BOOLP TRI-STATE)
       (INPUTS CLK EN SEL D Q)
       (LOADINGS 1 2 2 1 2)
       (OUT-DEPENDS (EN))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS Q)
       (STATE-TYPES . BOOLP)
       (STATES . LATCH)
       (GATES . 14)
       (PADS . 0)
       (PRIMITIVES . 3)
       (TRANSISTORS . 50)))
```

```
(netlist-database netlist T)
 '((M2 (DELAYS (OR ((146 (RANGE 2361 2441))
                    (57 (RANGE 2331 2411))
                    EN0)
                   ((146 (RANGE 2361 2441))
                    (57 (RANGE 2331 2411))
                    EN1)))
       (DRIVES 5)
       (INPUT-TYPES CLK BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP)
       (INPUTS CLK EN0 EN1 SEL0 SEL1 D0 D1)
       (LOADINGS 2 2 2 2 2 1 1)
       (OUT-DEPENDS (EN0 EN1))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS Q)
       (STATE-TYPES BOOLP BOOLP)
       (STATES OCC0 OCC1)
       (GATES . 28)
       (PADS . 0)
       (PRIMITIVES . 6)
       (TRANSISTORS . 100))
   (M1 (DELAYS ((146 (RANGE 1631 1711))
                (57 (RANGE 2046 2126))
                EN))
       (DRIVES 10)
       (INPUT-TYPES CLK BOOLP BOOLP BOOLP TRI-STATE)
       (INPUTS CLK EN SEL D Q)
       (LOADINGS 1 2 2 1 2)
       (OUT-DEPENDS (EN))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS Q)
       (STATE-TYPES . BOOLP)
       (STATES . LATCH)
       (GATES . 14)
       (PADS . 0)
       (PRIMITIVES . 3)
       (TRANSISTORS . 50)))
```

```
(setq netlist '((m2 (clk en0 en1 sel0 sel1 d0 d1 q) (q)
                    ((occ0 (q0) m1     (clk en0 sel0 d0 q))
                     (occ1 (q1) m1     (clk en1 sel1 d1 q))
                     (wire (q)  t-wire (q0 q1)))
                    (occ0 occ1))
                (m1 (clk en sel d q) (q)
                    ((qpullup (qb) pullup (q))
                     (mux   (b)    b-if  (sel d qb))
                     (latch (a an) fd1   (b clk))
                     (tbuf  (q)    t-buf (en a)))
                    latch)))
```

```
(top-level-predicate netlist)
 T
```

```
(netlist-database netlist F)
 '((M2 (DELAYS (OR ((146 459)
                    (57 785)
                    (RANGE 1318 1398)
                    EN0)
                   ((146 459)
                    (57 785)
                    (RANGE 1318 1398)
                    EN1)))
       (DRIVES 9)
       (INPUT-TYPES CLK BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP TRI-STATE)
       (INPUTS CLK EN0 EN1 SEL0 SEL1 D0 D1 Q)
       (LOADINGS 2 2 2 2 2 1 1 4)
       (OUT-DEPENDS (EN0 EN1))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS Q)
       (STATE-TYPES BOOLP BOOLP)
       (STATES OCC0 OCC1)
       (GATES . 28)
       (PADS . 0)
       (PRIMITIVES . 6)
       (TRANSISTORS . 100))
   (M1 (DELAYS ((146 313)
                (57 728)
                (RANGE 1318 1398)
                EN))
       (DRIVES 10)
       (INPUT-TYPES CLK BOOLP BOOLP BOOLP TRI-STATE)
       (INPUTS CLK EN SEL D Q)
       (LOADINGS 1 2 2 1 2)
       (OUT-DEPENDS (EN))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS Q)
       (STATE-TYPES . BOOLP)
       (STATES . LATCH)
       (GATES . 14)
       (PADS . 0)
       (PRIMITIVES . 3)
       (TRANSISTORS . 50)))
```

```
(netlist-database netlist T)
 '((M2 (DELAYS (OR ((146 (RANGE 1777 1857))
                    (57 (RANGE 2103 2183))
                    EN0)
                   ((146 (RANGE 1777 1857))
                    (57 (RANGE 2103 2183))
                    EN1)))
       (DRIVES 9)
       (INPUT-TYPES CLK BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP TRI-STATE)
       (INPUTS CLK EN0 EN1 SEL0 SEL1 D0 D1 Q)
       (LOADINGS 2 2 2 2 2 1 1 4)
       (OUT-DEPENDS (EN0 EN1))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS Q)
       (STATE-TYPES BOOLP BOOLP)
       (STATES OCC0 OCC1)
       (GATES . 28)
       (PADS . 0)
       (PRIMITIVES . 6)
       (TRANSISTORS . 100))
   (M1 (DELAYS ((146 (RANGE 1631 1711))
                (57 (RANGE 2046 2126))
                EN))
       (DRIVES 10)
       (INPUT-TYPES CLK BOOLP BOOLP BOOLP TRI-STATE)
       (INPUTS CLK EN SEL D Q)
       (LOADINGS 1 2 2 1 2)
       (OUT-DEPENDS (EN))
       (OUTPUT-TYPES TRI-STATE)
       (OUTPUTS Q)
       (STATE-TYPES . BOOLP)
       (STATES . LATCH)
       (GATES . 14)
       (PADS . 0)
       (PRIMITIVES . 3)
       (TRANSISTORS . 50)))


(setq args-if2 '((c) (v0 v1) (x0) (v0 w0) (y0 y1)))
; ?? (out-depends 0 'b-if2 args-if2 nil nil)




(setq args-16
      '((C-in) (a0) (a1) (a2) (a3) (a4) (a5) (a6) (a7) (a8) (a9) (a10)
        (a11) (a12) (a13) (a14) (a15) (b0) (b1) (b2) (b3) (b4) (b5) (b6) (b7)
        (b8) (b9) (b10) (b11) (b12) (b13) (b14) (b15) zero (mpg0) (mpg1)
        (mpg2) (mpg3) (mpg4) (mpg5) (mpg6) (op0) (op1) (op2) (op3)))
```

```
(setq arg-types-16
      '(boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp boolp boolp boolp boolp boolp boolp))


(setq netlist-16 (core-alu$netlist (make-tree 16)))


(netlist-syntax-ok netlist-16)

(lsi-netlist-syntax-ok (lisp-netlist netlist-16) 30 255)


(output-dependencies (index 'core-alu 16) args-16 netlist-16)


(netlist-type-table netlist-16)

(arg-types-okp (index 'core-alu 16) arg-types-16 netlist-16)


(netlist-loadings-and-drives netlist-16)

(netlist-properties netlist-16 '(delays) F)



(setq args-core32
      '((c)
        (a0)  (a1)  (a2)  (a3)  (a4)  (a5)  (a6)  (a7)  (a8)  (a9)  (a10)
        (a11) (a12) (a13) (a14) (a15) (a16) (a17) (a18) (a19) (a20) (a21)
        (a22) (a23) (a24) (a25) (a26) (a27) (a28) (a29) (a30) (a31)
        (b0)  (b1)  (b2)  (b3)  (b4)  (b5)  (b6)  (b7)  (b8)  (b9)  (b10)
        (b11) (b12) (b13) (b14) (b15) (b16) (b17) (b18) (b19) (b20) (b21)
        (b22) (b23) (b24) (b25) (b26) (b27) (b28) (b29) (b30) (b31)
        (zero)
        (mpg-0) (mpg-1) (mpg-2) (mpg-3) (mpg-4) (mpg-5) (mpg-6)
        (op-0) (op-1) (op-2) (op-3)))


(setq arg-types-core32
      '(boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp boolp boolp boolp boolp boolp boolp
        boolp boolp boolp boolp boolp))
```

```
(setq netlist-32 (core-alu$netlist (make-tree 32)))

(top-level-predicate netlist-32)
 T

(setq module-database (dependency-table netlist-32))


(output-dependencies (index 'core-alu 32) args-core32 netlist-32)

(netlist-syntax-ok netlist-32)

(lsi-netlist-syntax-ok (lisp-netlist netlist-32) 20 200)


(netlist-type-table netlist-32)

(arg-types-okp (index 'core-alu 32) arg-types-core32 netlist-32)


(netlist-loadings-and-drives netlist-32)

|#


#|


(setq ex-net '((m1 (in1 in2 clk) (out1 out2 out3 out4)
                   ((o1 (out1 out2 out3 out4)
                        fd11
                        (in1 in2 clk)))
                   o1)
               (fd11 (in1 in2 clk)
                     (out1 out2 out3 out4)
                     ((o1 (out1 out2) fd1 (in1 clk))
                      (o2 (out3 out4) fd1 (in2 clk)))
                     (o1 o2))))


(netlist-state-types ex-net)
 '((M1 (STATE-TYPES BOOLP BOOLP))
   (FD11 (STATE-TYPES BOOLP BOOLP)))


(state-type-requirement 0 'm1 ex-net)
 '(BOOLP BOOLP)


(state-type-requirement 0 'fd11 ex-net)
 '(BOOLP BOOLP)

; in following, first 4 T, remaining 6 F
```

```
(list (state-okp 'm1 (list f f) ex-net)
      (state-okp 'm1 (list f t) ex-net)
      (state-okp 'm1 (list t f) ex-net)
      (state-okp 'm1 (list t t) ex-net)
      (state-okp 'm1 (list t (x)) ex-net)
      (state-okp 'm1 (list (x) f) ex-net)
      (state-okp 'm1 (list t f t) ex-net)
      (state-okp 'm1 (list f) ex-net)
      (state-okp 'm1 nil ex-net)
      (state-okp 'm1 (x) ex-net))


(netlist-loadings-and-drives ex-net)
 '((M1 (LOADINGS 1 1 2)
       (DRIVES 10 10 10 10))
   (FD11 (LOADINGS 1 1 2)
         (DRIVES 10 10 10 10)))


(netlist-properties ex-net '(delays) F)

(netlist-properties ex-net '(delays) T)
 '((M1 (DELAYS ((147 1024) (55 1288))
               ((145 1432) (53 1447))
               ((147 1024) (55 1288))
               ((145 1432) (53 1447))))
   (FD11 (DELAYS ((147 1024) (55 1288))
                 ((145 1432) (53 1447))
                 ((147 1024) (55 1288))
                 ((145 1432) (53 1447)))))



(setq ex-net
      '((m1 (in1 in2 in3 in4 clk) (out1 out2 out3 out4)
            ((o1 (out1 w1) fd1 (in1 clk))
             (o2 (out2 w2) fd1 (in2 clk))
             (o3 (out3 w3) fd1 (in3 clk))
             (o4 (out4 w4) fd1 (in4 clk)))
            (o1 o2 o3 o4))))


(netlist-state-types ex-net)
 '((M1 (STATE-TYPES BOOLP BOOLP BOOLP BOOLP)))

; in following first 4 T, remaining 6 F
```

```
(list (state-okp 'm1 (list f f f f) ex-net)
      (state-okp 'm1 (list f t f t) ex-net)
      (state-okp 'm1 (list t f t f) ex-net)
      (state-okp 'm1 (list t t t t) ex-net)
      (state-okp 'm1 (list t f (x) t) ex-net)
      (state-okp 'm1 (list (x) t t t) ex-net)
      (state-okp 'm1 (list t f t f t) ex-net)
      (state-okp 'm1 (list f t f) ex-net)
      (state-okp 'm1 nil ex-net)
      (state-okp 'm1 (x) ex-net))


(netlist-loadings-and-drives ex-net)
 '((M1 (LOADINGS 1 1 1 1 4)
       (DRIVES 10 10 10 10)))


(netlist-properties ex-net '(delays) F)

(netlist-properties ex-net '(delays) T)
 '((M1 (DELAYS ((147 1024) (55 1288))
               ((147 1024) (55 1288))
               ((147 1024) (55 1288))
               ((147 1024) (55 1288)))))



(setq ex-net
      '((m1 (in1 in2 in3 in4 clk) (out1 out2)
            ((o1 (is1 w1) fd1 (in1 clk))
             (o2 (is2 w2) fd1 (in2 clk))
             (o3 (out1) b-and (is1 is2))
             (o4 (is3 w3) fd1 (in3 clk))
             (o5 (is4 w4) fd1 (in4 clk))
             (o6 (out2) b-and (is3 is4)))
            (o1 o2 o4 o5))))


(netlist-state-types ex-net)
 '((M1 (STATE-TYPES BOOLP BOOLP BOOLP BOOLP)))

; in following first 4 T, remaining 6 F

(list (state-okp 'm1 (list f f f f) ex-net)
      (state-okp 'm1 (list f t f t) ex-net)
      (state-okp 'm1 (list t f t f) ex-net)
      (state-okp 'm1 (list t t t t) ex-net)
      (state-okp 'm1 (list t f (x) t) ex-net)
      (state-okp 'm1 (list (x) t t t) ex-net)
      (state-okp 'm1 (list t f t f t) ex-net)
      (state-okp 'm1 (list f t f) ex-net)
      (state-okp 'm1 nil ex-net)
      (state-okp 'm1 (x) ex-net))
```

```
(netlist-loadings-and-drives ex-net)
 '((M1 (LOADINGS 1 1 1 1 4)
       (DRIVES 10 10)))


(netlist-properties ex-net '(delays) F)

(netlist-properties ex-net '(delays) T)
 '((M1 (DELAYS ((144 (RANGE 1593 1765))
                (54 (RANGE 1878 2050)))
               ((144 (RANGE 1593 1765))
                (54 (RANGE 1878 2050))))))



(setq ex-net
      '((m1
         (read-a0 read-a1 read-a2 read-a3
          write-b0 write-b1 write-b2 write-b3
          wen
          d0  d1  d2  d3  d4  d5  d6  d7
          d8  d9  d10 d11 d12 d13 d14 d15
          d16 d17 d18 d19 d20 d21 d22 d23
          d24 d25 d26 d27 d28 d29 d30 d31)
         (o0  o1  o2  o3  o4  o5  o6  o7
          o8  o9  o10 o11 o12 o13 o14 o15
          o16 o17 o18 o19 o20 o21 o22 o23
          o24 o25 o26 o27 o28 o29 o30 o31)
         ((o1
           (o0  o1  o2  o3  o4  o5  o6  o7
            o8  o9  o10 o11 o12 o13 o14 o15
            o16 o17 o18 o19 o20 o21 o22 o23
            o24 o25 o26 o27 o28 o29 o30 o31)
           dp-ram-16x32
           (read-a0 read-a1 read-a2 read-a3
            write-b0 write-b1 write-b2 write-b3
            wen
            d0  d1  d2  d3  d4  d5  d6  d7
            d8  d9  d10 d11 d12 d13 d14 d15
            d16 d17 d18 d19 d20 d21 d22 d23
            d24 d25 d26 d27 d28 d29 d30 d31)))
         o1)))


(netlist-state-types ex-net)
 '((M1 (STATE-TYPES ADDRESSED-STATE 4
                    (RAM '(BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP
                                 BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP
                                 BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP
                                 BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP
                                 BOOLP BOOLP BOOLP)))))

; in following, first 2 T, remaining 5 F
```

```
(list (state-okp 'm1 (make-ram-state (make-tree 16) 32 t) ex-net)
      (state-okp 'm1 (make-ram-state (make-tree 16) 32 f) ex-net)
      (state-okp 'm1 (make-ram-state (make-tree 8) 32 f) ex-net)
      (state-okp 'm1 (make-ram-state (make-tree 16) 8 f) ex-net)
      (state-okp 'm1 (list (make-ram-state (make-tree 16) 32 t)) ex-net)
      (state-okp 'm1 (x) ex-net)
      (state-okp 'm1 nil ex-net))
```

```
(netlist-loadings-and-drives ex-net)
 '((M1 (LOADINGS 2 2 2 2 2 2 2 2 4 1 1 1 1 1 1 1 1 1 1 1 1 1
                 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
       (DRIVES 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
               2)))
```

```
(netlist-properties ex-net '(delays) F)
```

```
(netlist-properties ex-net '(delays) T)
 '((M1 (DELAYS ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
```

```
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3))))
```

```
(setq ex-net
      '((m1
         (clk
          read-a0 read-a1 read-a2 read-a3
          write-b0 write-b1 write-b2 write-b3
          wen
          d0  d1  d2  d3  d4  d5  d6  d7
          d8  d9  d10 d11 d12 d13 d14 d15
          d16 d17 d18 d19 d20 d21 d22 d23
          d24 d25 d26 d27 d28 d29 d30 d31)
         (o0  o1  o2  o3  o4  o5  o6  o7
          o8  o9  o10 o11 o12 o13 o14 o15
          o16 o17 o18 o19 o20 o21 o22 o23
          o24 o25 o26 o27 o28 o29 o30 o31)
         ((o0 (is w2) fd1 (d0 clk))
          (o1
           (o0  o1  o2  o3  o4  o5  o6  o7
            o8  o9  o10 o11 o12 o13 o14 o15
            o16 o17 o18 o19 o20 o21 o22 o23
            o24 o25 o26 o27 o28 o29 o30 o31)
           dp-ram-16x32
           (read-a0 read-a1 read-a2 read-a3
            write-b0 write-b1 write-b2 write-b3
            wen
            is  d1  d2  d3  d4  d5  d6  d7
            d8  d9  d10 d11 d12 d13 d14 d15
            d16 d17 d18 d19 d20 d21 d22 d23
            d24 d25 d26 d27 d28 d29 d30 d31)))
         (o0 o1))))


(netlist-state-types ex-net)
 '((M1 (STATE-TYPES BOOLP
                    (ADDRESSED-STATE 4
                                     (RAM '(BOOLP BOOLP BOOLP BOOLP BOOLP
                                                  BOOLP BOOLP BOOLP BOOLP
                                                  BOOLP BOOLP BOOLP BOOLP
                                                  BOOLP BOOLP BOOLP BOOLP
                                                  BOOLP BOOLP BOOLP BOOLP
                                                  BOOLP BOOLP BOOLP BOOLP
                                                  BOOLP BOOLP BOOLP BOOLP
                                                  BOOLP BOOLP BOOLP))))))

; in following, first 4 T, remaining 7 F
```

```
(list (state-okp 'm1 (list f (make-ram-state (make-tree 16) 32 f)) ex-net)
      (state-okp 'm1 (list f (make-ram-state (make-tree 16) 32 t)) ex-net)
      (state-okp 'm1 (list t (make-ram-state (make-tree 16) 32 f)) ex-net)
      (state-okp 'm1 (list t (make-ram-state (make-tree 16) 32 t)) ex-net)
      (state-okp 'm1 (list t (make-ram-state (make-tree 15) 32 t)) ex-net)
      (state-okp 'm1 (list t (make-ram-state (make-tree 16) 33 t)) ex-net)
      (state-okp 'm1 (list f (make-ram-state (make-tree 16) 32 t) t) ex-net)
      (state-okp 'm1 (list (make-ram-state (make-tree 16) 32 t)) ex-net)
      (state-okp 'm1 (list t) ex-net)
      (state-okp 'm1 (x) ex-net)
      (state-okp 'm1 nil ex-net))
```

```
(netlist-loadings-and-drives ex-net)
 '((M1 (LOADINGS 1
                 2 2 2 2 2 2 2 2 4 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                 1 1 1 1 1 1 1 1 1 1 1)
       (DRIVES 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
               2)))
```

```
(netlist-properties ex-net '(delays) F)
```

```
(netlist-properties ex-net '(delays) T)
 '((M1 (DELAYS ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
```

```
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3))))

|#
```

```
(defn test-net2 ()
      '((m1 (read-a0 read-a1 read-a2 read-a3
             write-b0 write-b1 write-b2 write-b3
             in0
             d0  d1  d2  d3  d4  d5  d6  d7
             d8  d9  d10 d11 d12 d13 d14 d15
             d16 d17 d18 d19 d20 d21 d22 d23
             d24 d25 d26 d27 d28 d29 d30 d31
             clk)
            (out0)
            ((o1 (is1) m5 (in0))
             (o2 (is2) m3 (is1 clk))
             (o3 (is3) m4 (in0 is2))
             (o4 (wen) m8 (clk is1 is3))
             (o5
              (o0  o1  o2  o3  o4  o5  o6  o7
               o8  o9  o10 o11 o12 o13 o14 o15
               o16 o17 o18 o19 o20 o21 o22 o23
               o24 o25 o26 o27 o28 o29 o30 o31 out0)
              m2
              (read-a0 read-a1 read-a2 read-a3
               write-b0 write-b1 write-b2 write-b3
               wen
               d0  d1  d2  d3  d4  d5  d6  d7
               d8  d9  d10 d11 d12 d13 d14 d15
               d16 d17 d18 d19 d20 d21 d22 d23
               d24 d25 d26 d27 d28 d29 d30 d31
               clk)))
            (o2 o3 o5))
        (m2 (read-a0 read-a1 read-a2 read-a3
             write-b0 write-b1 write-b2 write-b3
             wen
             d0  d1  d2  d3  d4  d5  d6  d7
             d8  d9  d10 d11 d12 d13 d14 d15
             d16 d17 d18 d19 d20 d21 d22 d23
             d24 d25 d26 d27 d28 d29 d30 d31
             clk)
            (o0  o1  o2  o3  o4  o5  o6  o7
             o8  o9  o10 o11 o12 o13 o14 o15
             o16 o17 o18 o19 o20 o21 o22 o23
             o24 o25 o26 o27 o28 o29 o30 o31 out0)
            ((o1
              (o0  o1  o2  o3  o4  o5  o6  o7
               o8  o9  o10 o11 o12 o13 o14 o15
               o16 o17 o18 o19 o20 o21 o22 o23
               o24 o25 o26 o27 o28 o29 o30 o31)
              m6
              (read-a0 read-a1 read-a2 read-a3
               write-b0 write-b1 write-b2 write-b3
               wen
               d0  d1  d2  d3  d4  d5  d6  d7
               d8  d9  d10 d11 d12 d13 d14 d15
               d16 d17 d18 d19 d20 d21 d22 d23
               d24 d25 d26 d27 d28 d29 d30 d31))
```

```
         (o2 (out0) m3 (o31 clk)))
        (o1 o2))
    (m3 (in0 clk) (out0)
        ((o1 (out0) m7 (clk in0)))
        o1)
    (m4 (in1 in2) (out0)
        ((o1 (is1) m5 (in1))
         (o2 (is2 is3 is4 is5) fd11 (is1 in2 clk))
         (o3 (out0) b-and4 (is2 is3 is4 is5)))
        o2)
    (m5 (in0) (out0)
        ((o1 (out0) b-not (in0)))
        nil)
    (m6 (read-a0 read-a1 read-a2 read-a3
         write-b0 write-b1 write-b2 write-b3
         wen
         d0  d1  d2  d3  d4  d5  d6  d7
         d8  d9  d10 d11 d12 d13 d14 d15
         d16 d17 d18 d19 d20 d21 d22 d23
         d24 d25 d26 d27 d28 d29 d30 d31)
        (o0  o1  o2  o3  o4  o5  o6  o7
         o8  o9  o10 o11 o12 o13 o14 o15
         o16 o17 o18 o19 o20 o21 o22 o23
         o24 o25 o26 o27 o28 o29 o30 o31)
        ((o-1
          (o0  o1  o2  o3  o4  o5  o6  o7
           o8  o9  o10 o11 o12 o13 o14 o15
           o16 o17 o18 o19 o20 o21 o22 o23
           o24 o25 o26 o27 o28 o29 o30 o31)
          dp-ram-16x32
          (read-a0 read-a1 read-a2 read-a3
           write-b0 write-b1 write-b2 write-b3
           wen
           d0  d1  d2  d3  d4  d5  d6  d7
           d8  d9  d10 d11 d12 d13 d14 d15
           d16 d17 d18 d19 d20 d21 d22 d23
           d24 d25 d26 d27 d28 d29 d30 d31)))
        o-1)
    (m7 (clk in0) (out0)
        ((o1 (out0 w) fd1 (in0 clk)))
        o1)
    (m8 (clk in1 in2) (o)
        ((oc0 (is1) vdd ())
         (oc1 (o) ram-enable-circuit (clk is1 in1 in2)))
        nil)
    (fd11 (in1 in2 clk)
          (out1 out2 out3 out4)
          ((o1 (out1 out2) fd1 (in1 clk))
           (o2 (out3 out4) fd1 (in2 clk)))
          (o1 o2))))

#|
```

```
(setq state
      (list t (list t f) (list (make-ram-state (make-tree 16) 32 t) f)))


(setq ex-net (test-net2))


(netlist-state-types ex-net)
 '((M1 (STATE-TYPES BOOLP
                    (BOOLP BOOLP)
                    ((ADDRESSED-STATE 4
                                      (RAM '(BOOLP BOOLP BOOLP BOOLP BOOLP
                                                   BOOLP BOOLP BOOLP BOOLP
                                                   BOOLP BOOLP BOOLP BOOLP
                                                   BOOLP BOOLP BOOLP BOOLP
                                                   BOOLP BOOLP BOOLP BOOLP
                                                   BOOLP BOOLP BOOLP BOOLP
                                                   BOOLP BOOLP BOOLP BOOLP
                                                   BOOLP BOOLP BOOLP)))
                     BOOLP)))
   (M2 (STATE-TYPES (ADDRESSED-STATE 4
                                     (RAM '(BOOLP BOOLP BOOLP BOOLP BOOLP
                                                  BOOLP BOOLP BOOLP BOOLP
                                                  BOOLP BOOLP BOOLP BOOLP
                                                  BOOLP BOOLP BOOLP BOOLP
                                                  BOOLP BOOLP BOOLP BOOLP
                                                  BOOLP BOOLP BOOLP BOOLP
                                                  BOOLP BOOLP BOOLP BOOLP
                                                  BOOLP BOOLP BOOLP)))
                    BOOLP))
   (M3 (STATE-TYPES . BOOLP))
   (M4 (STATE-TYPES BOOLP BOOLP))
   (M6 (STATE-TYPES ADDRESSED-STATE 4
                    (RAM '(BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP
                                 BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP
                                 BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP
                                 BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP BOOLP
                                 BOOLP BOOLP BOOLP))))
   (M7 (STATE-TYPES . BOOLP))
   (FD11 (STATE-TYPES BOOLP BOOLP)))


(state-okp 'm1 state ex-net) ; repeat with flaws in state
```

```
(netlist-loadings-and-drives ex-net)
 '((M1 (LOADINGS 2 2 2 2 2 2 2 2 4 1 1 1 1 1 1 1 1 1 1 1 1
                 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3)
       (DRIVES 10))
   (M2 (LOADINGS 2 2 2 2 2 2 2 2 4 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                 1 1 1 1 1 1 1 1 1 1 1 1)
       (DRIVES 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
               1 10))
   (M3 (LOADINGS 1 1) (DRIVES 10))
   (M4 (LOADINGS 2 1) (DRIVES 10))
   (M5 (LOADINGS 2) (DRIVES 10))
   (M6 (LOADINGS 2 2 2 2 2 2 2 2 4 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                 1 1 1 1 1 1 1 1 1 1 1)
       (DRIVES 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
               2))
   (M7 (LOADINGS 1 1) (DRIVES 10))
   (M8 (LOADINGS 1 2 2) (DRIVES 10))
   (FD11 (LOADINGS 1 1 2)
         (DRIVES 10 10 10 10)))
```

```
(netlist-properties ex-net '(delays) F)
```

```
(netlist-properties ex-net '(delays) T)
 '((M1 (DELAYS ((147 1024) (55 1288))))
   (M2 (DELAYS ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
                (21 7500)
                READ-A0 READ-A1 READ-A2 READ-A3)
               ((35 7500)
```

```
                 (21 7500)
                 READ-A0 READ-A1 READ-A2 READ-A3)
                ((35 7500)
                 (21 7500)
                 READ-A0 READ-A1 READ-A2 READ-A3)
                ((35 7500)
                 (21 7500)
                 READ-A0 READ-A1 READ-A2 READ-A3)
                ((35 7500)
                 (21 7500)
                 READ-A0 READ-A1 READ-A2 READ-A3)
                ((35 7500)
                 (21 7500)
                 READ-A0 READ-A1 READ-A2 READ-A3)
                ((35 7500)
                 (21 7500)
                 READ-A0 READ-A1 READ-A2 READ-A3)
                ((35 7500)
                 (21 7500)
                 READ-A0 READ-A1 READ-A2 READ-A3)
                ((35 7500)
                 (21 7500)
                 READ-A0 READ-A1 READ-A2 READ-A3)
                ((35 7500)
                 (21 7500)
                 READ-A0 READ-A1 READ-A2 READ-A3)
                ((35 7500)
                 (21 7500)
                 READ-A0 READ-A1 READ-A2 READ-A3)
                ((35 7500)
                 (21 7500)
                 READ-A0 READ-A1 READ-A2 READ-A3)
                ((35 7500)
                 (21 7500)
                 READ-A0 READ-A1 READ-A2 READ-A3)
                ((35 7500)
                 (21 7500)
                 READ-A0 READ-A1 READ-A2 READ-A3)
                ((35 7500)
                 (21 7500)
                 READ-A0 READ-A1 READ-A2 READ-A3)
                ((35 7535)
                 (21 7521)
                 READ-A0 READ-A1 READ-A2 READ-A3)
                ((147 1024) (55 1288))))
   (M3 (DELAYS ((147 1024) (55 1288))))
   (M4 (DELAYS ((149 (RANGE 2434 2511))
                 (60 (RANGE 2370 2447)))))
   (M5 (DELAYS ((70 235) (57 198) IN0)))
   (M6 (DELAYS ((35 7500)
                 (21 7500)
                 READ-A0 READ-A1 READ-A2 READ-A3)
                ((35 7500)
                 (21 7500)
                 READ-A0 READ-A1 READ-A2 READ-A3)
```

```
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
 (21 7500)
 READ-A0 READ-A1 READ-A2 READ-A3)
((35 7500)
```

```
              (21 7500)
              READ-A0 READ-A1 READ-A2 READ-A3)
             ((35 7500)
              (21 7500)
              READ-A0 READ-A1 READ-A2 READ-A3)
             ((35 7500)
              (21 7500)
              READ-A0 READ-A1 READ-A2 READ-A3)
             ((35 7500)
              (21 7500)
              READ-A0 READ-A1 READ-A2 READ-A3)
             ((35 7500)
              (21 7500)
              READ-A0 READ-A1 READ-A2 READ-A3)
             ((35 7500)
              (21 7500)
              READ-A0 READ-A1 READ-A2 READ-A3)
             ((35 7500)
              (21 7500)
              READ-A0 READ-A1 READ-A2 READ-A3)
             ((35 7500)
              (21 7500)
              READ-A0 READ-A1 READ-A2 READ-A3)
             ((35 7500)
              (21 7500)
              READ-A0 READ-A1 READ-A2 READ-A3)
             ((35 7500)
              (21 7500)
              READ-A0 READ-A1 READ-A2 READ-A3)
             ((35 7500)
              (21 7500)
              READ-A0 READ-A1 READ-A2 READ-A3)
             ((35 7500)
              (21 7500)
              READ-A0 READ-A1 READ-A2 READ-A3)))
   (M7 (DELAYS ((147 1024) (55 1288))))
   (M8 (DELAYS ((68 12000) (54 12000) CLK IN1 IN2)))
   (FD11 (DELAYS ((147 1024) (55 1288))
               ((145 1432) (53 1447))
               ((147 1024) (55 1288))
               ((145 1432) (53 1447)))))
|#
```

```
(defn test-net3 ()
      '((m1 (read-a0 read-a1 read-a2 read-a3
             write-b0 write-b1 write-b2 write-b3
             in0
             d0  d1  d2  d3  d4  d5  d6  d7
             d8  d9  d10 d11 d12 d13 d14 d15
             d16 d17 d18 d19 d20 d21 d22 d23
             d24 d25 d26 d27 d28 d29 d30 d31
             clk)
            (out0)
            ((o1 (is1) m5 (in0))
             (o2 (is2) m3 (is1 clk))
             (o3 (is3) m4 (in0 is2))
             (o4 (wen) m8 (clk is1 is3))
             (o5
              (o0  o1  o2  o3  o4  o5  o6  o7
               o8  o9  o10 o11 o12 o13 o14 o15
               o16 o17 o18 o19 o20 o21 o22 o23
               o24 o25 o26 o27 o28 o29 o30 o31 out0)
              m2
              (read-a0 read-a1 read-a2 read-a3
               write-b0 write-b1 write-b2 write-b3
               wen
               d0  d1  d2  d3  d4  d5  d6  d7
               d8  d9  d10 d11 d12 d13 d14 d15
               d16 d17 d18 d19 d20 d21 d22 d23
               d24 d25 d26 d27 d28 d29 d30 d31
               clk)))
            (o2 o3 o4 o5))
        (m2 (read-a0 read-a1 read-a2 read-a3
             write-b0 write-b1 write-b2 write-b3
             wen
             d0  d1  d2  d3  d4  d5  d6  d7
             d8  d9  d10 d11 d12 d13 d14 d15
             d16 d17 d18 d19 d20 d21 d22 d23
             d24 d25 d26 d27 d28 d29 d30 d31
             clk)
            (o0  o1  o2  o3  o4  o5  o6  o7
             o8  o9  o10 o11 o12 o13 o14 o15
             o16 o17 o18 o19 o20 o21 o22 o23
             o24 o25 o26 o27 o28 o29 o30 o31 out0)
            ((o1
              (o0  o1  o2  o3  o4  o5  o6  o7
               o8  o9  o10 o11 o12 o13 o14 o15
               o16 o17 o18 o19 o20 o21 o22 o23
               o24 o25 o26 o27 o28 o29 o30 o31)
              m6
              (read-a0 read-a1 read-a2 read-a3
               write-b0 write-b1 write-b2 write-b3
               wen
               d0  d1  d2  d3  d4  d5  d6  d7
               d8  d9  d10 d11 d12 d13 d14 d15
               d16 d17 d18 d19 d20 d21 d22 d23
               d24 d25 d26 d27 d28 d29 d30 d31))
```

```
             (o2 (out0) m3 (o31 clk)))
            o2)
        (m3 (in0 clk) (out0)
            ((o1 (out0) m7 (clk in0)))
            nil)
        (m4 (in1 in2) (out0)
            ((o1 (is1) m5 (in1))
             (o2 (is2 is3 is4 is5) fd11 (is1 in2 clk))
             (o3 (out0) b-and4 (is2 is3 is4 is5)))
            o2)
        (m5 (in0) (out0)
            ((o1 (out0) b-not (in0)))
            o1)
        (m6 (read-a0 read-a1 read-a2 read-a3
             write-b0 write-b1 write-b2 write-b3
             wen
             d0  d1  d2  d3  d4  d5  d6  d7
             d8  d9  d10 d11 d12 d13 d14 d15
             d16 d17 d18 d19 d20 d21 d22 d23
             d24 d25 d26 d27 d28 d29 d30 d31)
            (o0  o1  o2  o3  o4  o5  o6  o7
             o8  o9  o10 o11 o12 o13 o14 o15
             o16 o17 o18 o19 o20 o21 o22 o23
             o24 o25 o26 o27 o28 o29 o30 o31)
            ((o-1
              (o0  o1  o2  o3  o4  o5  o6  o7
               o8  o9  o10 o11 o12 o13 o14 o15
               o16 o17 o18 o19 o20 o21 o22 o23
               o24 o25 o26 o27 o28 o29 o30 o31)
              dp-ram-16x32
              (read-a0 read-a1 read-a2 read-a3
               write-b0 write-b1 write-b2 write-b3
               wen
               d0  d1  d2  d3  d4  d5  d6  d7
               d8  d9  d10 d11 d12 d13 d14 d15
               d16 d17 d18 d19 d20 d21 d22 d23
               d24 d25 d26 d27 d28 d29 d30 d31)))
            o1)
        (m7 (clk in0) (out0)
            ((o1 (out0 w) fd1 (in0 clk)))
            o1)
        (m8 (clk in1 in2) (o)
            ((oc0 (is1) vdd ())
             (oc1 (o) ram-enable-circuit (clk is1 in1 in2)))
            nil)
        (fd11 (in1 in2 clk)
              (out1 out2 out3 out4)
              ((o1 (out1 out2) fd1 (in1 clk))
               (o2 (out3 out4) fd1 (in2 clk)))
              (o1 o2))))

#|
```

```
(setq ex-net (test-net3))


(netlist-state-types ex-net)
 (NET-ERROR 'NETLIST-ERRORS
          (LIST (NET-ERROR '(MODULE M1)
                           (LIST (NET-ERROR 'BAD-STATE-TYPES
                                            '((MODULE-STATES (O2 O3 O4 O5))
                                              (COMPUTED-STATES (O5 O3 O1))))))
                (NET-ERROR '(MODULE M2)
                           (LIST (NET-ERROR 'BAD-STATE-TYPES
                                            '((MODULE-STATES O2)
                                              (COMPUTED-STATES (O1))))))
                (NET-ERROR '(MODULE M3)
                           (LIST (NET-ERROR 'BAD-STATE-TYPES
                                            '((MODULE-STATES NIL)
                                              (COMPUTED-STATES (O1))))))
                (NET-ERROR '(MODULE M4)
                           (LIST (NET-ERROR 'BAD-STATE-TYPES
                                            '((MODULE-STATES O2)
                                              (COMPUTED-STATES (O2 O1))))))
                (NET-ERROR '(MODULE M5)
                           (LIST (NET-ERROR 'BAD-STATE-TYPES
                                            '((MODULE-STATES O1)
                                              (COMPUTED-STATES NIL)))))
                (NET-ERROR '(MODULE M6)
                           (LIST (NET-ERROR 'BAD-STATE-TYPES
                                            '((MODULE-STATES O1)
                                              (COMPUTED-STATES (O-1))))))))



(defn list-set-equal (x y)
  (if (or (nlistp x) (nlistp y))
      (and (nlistp x) (nlistp y))
      (if (set-equal (car x) (car y))
          (list-set-equal (cdr x) (cdr y))
          f)))

|#
#|

(setq ex-net1
      '((m1 (in0) (out1 out2)
            ((o1 (out1) b-not (in0))
             (o2 (out2) id (out1)))
            nil)))


(netlist-loadings-and-drives ex-net1)
 '((M1 (LOADINGS 2) (DRIVES 10 OUT1)))


(netlist-properties ex-net1 '(delays) F)
```

```
(netlist-properties ex-net1 '(delays) T)
 '((M1 (DELAYS ((70 235) (57 198) IN0)
              OUT1)))



(setq ex-net2
      '((m1 (in1 in2) (out1 out2)
            ((o1 (is1) id (in1))
             (o2 (out1) b-and (is1 in2))
             (o3 (out2) b-or (is1 in2)))
            nil)))


(netlist-loadings-and-drives ex-net2)
 '((M1 (LOADINGS 2 2) (DRIVES 10 10)))


(netlist-properties ex-net2 '(delays) F)

(netlist-properties ex-net2 '(delays) T)
 '((M1 (DELAYS ((144 422) (54 707) IN1 IN2)
              ((143 332) (58 819) IN1 IN2))))



(setq ex-net3
      '((m1 (in0) (out1 out2)
            ((o1 (io1) id (in0))
             (o2 (out1) id (io1))
             (o3 (out2) id (io1)))
            nil)))


(netlist-loadings-and-drives ex-net3)
 '((M1 (LOADINGS 0) (DRIVES IN0 IN0)))


(netlist-properties ex-net3 '(delays) F)

(netlist-properties ex-net3 '(delays) T)
 '((M1 (DELAYS IN0 IN0)))



(setq ex-net4
      '((m1 (in0) (out1 out2)
            ((o1 (out1) id (in0))
             (o2 (out2) id (out1)))
            nil)))
```

```
(netlist-loadings-and-drives ex-net4)
 '((M1 (LOADINGS 0) (DRIVES IN0 IN0)))


(netlist-properties ex-net4 '(delays) F)

(netlist-properties ex-net4 '(delays) T)
 '((M1 (DELAYS IN0 IN0)))



(setq ex-net5
      '((m1 (in0) (out1 out2)
            ((o1 (out1) id (in0))
             (o2 (out2) id (in0)))
            nil)))


(netlist-loadings-and-drives ex-net5)
 '((M1 (LOADINGS 0) (DRIVES IN0 IN0)))


(netlist-properties ex-net5 '(delays) F)

(netlist-properties ex-net5 '(delays) T)
 '((M1 (DELAYS IN0 IN0)))



(setq ex-net6
      '((m1 () (out0)
            ((o1 (out0) vdd nil))
            nil)))


(netlist-loadings-and-drives ex-net6)
 '((M1 (LOADINGS) (DRIVES 50)))


(netlist-properties ex-net6 '(delays) F)

(netlist-properties ex-net6 '(delays) T)
 '((M1 (DELAYS ((0 0) (0 0)))))



(setq ex-net7
      '((m1 () (out0)
            ((o1 (out0) vss nil))
            nil)))
```

```
(netlist-loadings-and-drives ex-net7)
 '((M1 (LOADINGS) (DRIVES 50)))


(netlist-properties ex-net7 '(delays) F)

(netlist-properties ex-net7 '(delays) T)
 '((M1 (DELAYS ((0 0) (0 0)))))



(setq ex-net8
      '((m1 (a b) (c d)
            ((o1 (c) b-and (a b))
             (o2 (d) id (c)))
            nil)))


(netlist-loadings-and-drives ex-net8)
 '((M1 (LOADINGS 1 1) (DRIVES 10 C)))


(netlist-properties ex-net8 '(delays) F)

(netlist-properties ex-net8 '(delays) T)
 '((M1 (DELAYS ((144 422) (54 707) A B) C)))



(setq ex-net9
      '((m1 (in0) (out1 out2 out3 out4)
            ((o1 (out1) id (in0))
             (o2 (out2) b-not (out1))
             (o3 (out3) id (in0))
             (o4 (out4) b-not (in0)))
            nil)))


(netlist-loadings-and-drives ex-net9)
 '((M1 (LOADINGS 4)
       (DRIVES IN0 10 IN0 10)))


(netlist-properties ex-net9 '(delays) F)

(netlist-properties ex-net9 '(delays) T)
 '((M1 (DELAYS IN0
               ((70 235) (57 198) IN0)
               IN0
               ((70 235) (57 198) IN0))))
```

```
(setq ex-net10
      '((m1 (in0) (out0)
            ((o1 (a) id (in0))
             (o2 (b) id (in0))
             (o3 (out0) b-and (a b)))
            nil)))


(netlist-loadings-and-drives ex-net10)
 '((M1 (LOADINGS 2) (DRIVES 10)))


(netlist-properties ex-net10 '(delays) F)

(netlist-properties ex-net10 '(delays) T)
 '((M1 (DELAYS ((144 422) (54 707) IN0))))



(setq ex-net11
      '((m1 (in0) (out0)
            ((o1 (a) id (in0))
             (o2 (out0) b-and (in0 a)))
            nil)))


(netlist-loadings-and-drives ex-net11)
 '((M1 (LOADINGS 2) (DRIVES 10)))


(netlist-properties ex-net11 '(delays) F)

(netlist-properties ex-net11 '(delays) T)
 '((M1 (DELAYS ((144 422) (54 707) IN0))))



(setq ex-net12
      '((m1 (in0 in1 in2) (out0 out1 out2 out3)
            ((o1 (out1) id (in1))
             (o2 (out0) b-or (in0 out1))
             (o3 (out2) b-not (in1))
             (o4 (out3) b-and (in1 in2)))
            nil)))


(netlist-loadings-and-drives ex-net12)
 '((M1 (LOADINGS 1 4 1)
       (DRIVES 10 IN1 10 10)))


(netlist-properties ex-net12 '(delays) F)
```

```
(netlist-properties ex-net12 '(delays) T)
 '((M1 (DELAYS ((143 332) (58 819) IN0 IN1)
               IN1
               ((70 235) (57 198) IN1)
               ((144 422) (54 707) IN1 IN2))))



(setq ex-net13
      '((m1 (in0 in1 in2) (out0 out1 out2 out3)
            ((o1 (out1) id (in1))
             (o2 (out0) b-or (in0 out1))
             (o3 (out2) b-not (out1))
             (o4 (out3) b-and (out1 in2)))
            nil)))


(netlist-loadings-and-drives ex-net13)
 '((M1 (LOADINGS 1 4 1)
       (DRIVES 10 IN1 10 10)))


(netlist-properties ex-net13 '(delays) F)

(netlist-properties ex-net13 '(delays) T)
 '((M1 (DELAYS ((143 332) (58 819) IN0 IN1)
               IN1
               ((70 235) (57 198) IN1)
               ((144 422) (54 707) IN1 IN2))))



(setq ex-net14
      '((m1 (in0 clk) (out1 out2 out3)
            ((o1 (out1) b-not (in0))
             (o2 (is1) id (out1))
             (o3 (out2 w1) fd1 (is1 clk))
             (o4 (out3 w2) b-nbuf (is1)))
            o3)))


(netlist-loadings-and-drives ex-net14)
 '((M1 (LOADINGS 2 1) (DRIVES 8 10 9)))


(netlist-properties ex-net14 '(delays) F)

(netlist-properties ex-net14 '(delays) T)
 '((M1 (DELAYS ((70 375) (57 312) IN0)
               ((147 1024) (55 1288))
               ((142 447) (57 213) OUT1))))
```

```
(setq ex-net15
      '((m1 (in1 in2) (out1 out2)
            ((o1 (is1) b-and (in1 in2))
             (o2 (out1) b-not-b4ip (is1))
             (o3 (out2) b-not (is1)))
            nil)))


(netlist-loadings-and-drives ex-net15)
 '((M1 (LOADINGS 1 1) (DRIVES 128 10)))


(netlist-properties ex-net15 '(delays) F)
 '((M1 (DELAYS ((17 333)
                (12 124)
                (RANGE 1247 1862 IN1 IN2))
               ((70 235)
                (57 198)
                (RANGE 1247 1862 IN1 IN2)))))


(netlist-properties ex-net15 '(delays) T)
 '((M1 (DELAYS ((17 (RANGE 1580 2195))
                (12 (RANGE 1371 1986))
                IN1 IN2)
               ((70 (RANGE 1482 2097))
                (57 (RANGE 1445 2060))
                IN1 IN2))))



(setq ex-net16
      '((m1 (in1 in2) (out1 out2 out3)
            ((o1 (out1) b-and (in1 in2))
             (o2 (out2) b-not-b4ip (out1))
             (o3 (out3) b-not (out1)))
            nil)))


(netlist-loadings-and-drives ex-net16)
 '((M1 (LOADINGS 1 1) (DRIVES 0 128 10)))


(netlist-properties ex-net16 '(delays) F)

(netlist-properties ex-net16 '(delays) T)
 '((M1 (DELAYS ((144 1862) (54 1247) IN1 IN2)
               ((17 333) (12 124) OUT1)
               ((70 235) (57 198) OUT1))))
```

```
(setq ex-net17
      '((m1 (in1 in2) (out1 out2)
            ((o1 (is1) b-and (in1 in2))
             (o2 (out1) b-not-b4ip (is1))
             (o3 (out2) b-not-ivap (is1)))
            nil)))


(netlist-loadings-and-drives ex-net17)
 (NET-ERROR 'NETLIST-ERRORS
  (LIST
   (NET-ERROR
           '(MODULE M1)
           (LIST (NET-ERROR 'BAD-DRIVES
                            (LIST (NET-ERROR '(SIGNAL IS1)
                                             '((DRIVE 10) (LOADING 11)))))))))


(netlist-properties ex-net17 '(delays) F)

(netlist-properties ex-net17 '(delays) T)
 ; the drive error above


(setq ex-net18
      '((m1 (in1 in2) (out1 out2 out3)
            ((o1 (out1) b-and (in1 in2))
             (o2 (out2) b-not-b4ip (out1))
             (o3 (out3) b-not-ivap (out1)))
            nil)))


(netlist-loadings-and-drives ex-net18)
 (NET-ERROR 'NETLIST-ERRORS
  (LIST
   (NET-ERROR
           '(MODULE M1)
           (LIST (NET-ERROR 'BAD-DRIVES
                            (LIST (NET-ERROR '(SIGNAL OUT1)
                                             '((DRIVE 10) (LOADING 11)))))))))


(netlist-properties ex-net18 '(delays) F)

(netlist-properties ex-net18 '(delays) T)
; the drive error above


(setq ex-net19
      '((m1 (a b c d) (x)
            ((o1 (x) b-and (a b)))
            nil)))
```

```
(netlist-loadings-and-drives ex-net19)
 '((M1 (LOADINGS 1 1 0 0) (DRIVES 10)))


(netlist-properties ex-net19 '(delays) F)

(netlist-properties ex-net19 '(delays) T)
 '((M1 (DELAYS ((144 422) (54 707) A B))))



(setq ex-net20
      '((m1 (io a en pi b) (io x y z)
            ((o0 (is0) t-buf (en a))
             (o1 (is1) t-wire (b is0))
             (o2 (io zi po) ttl-bidirect (io a en pi))
             (o3 (x) pullup (io))
             (o4 (y) b-not (x))
             (o5 (is3) pullup (is1))
             (o6 (z) b-and (x is3)))
            nil)))


(netlist-properties ex-net20 '(delays) F)

(netlist-properties ex-net20 '(delays) T)
 (LIST (LIST 'M1
             (LIST 'DELAYS
                   '(((61 PS-PF) 4073)
                     ((41 PS-PF) 3893)
                     EN A)
                   (IO-OUT 'IO)
                   (LIST '(70 235)
                         '(57 198)
                         (IO-OUT 'IO))
                   (LIST '(144 422)
                         '(54 707)
                         (IO-OUT 'IO)
                         '(OR B (RANGE 751 899 A EN))))))
|#

#|

(setq ex-net
      '((m1 (a) (b c d)
            ((o1 (b) id (a))
             (o2 (c) id (a))
             (o3 (d) id (a)))
            nil)))


(netlist-loadings-and-drives ex-net)
 '((M1 (LOADINGS 0) (DRIVES A A A)))
```

```
(netlist-properties ex-net '(delays) F)

(netlist-properties ex-net '(delays) T)
 '((M1 (DELAYS A A A)))



(setq ex-net
      '((m1 (a) (b c d)
            ((o1 (b) id (a))
             (o2 (c) id (b))
             (o3 (d) id (c)))
            nil)))


(netlist-loadings-and-drives ex-net)
 '((M1 (LOADINGS 0) (DRIVES A A A)))


(netlist-properties ex-net '(delays) F)

(netlist-properties ex-net '(delays) T)
 '((M1 (DELAYS A A A)))



(setq ex-net
      '((m1 (a) (b c d)
            ((o1 (b) b-not (a))
             (o2 (c) id (a))
             (o3 (d) id (a)))
            nil)))


(netlist-loadings-and-drives ex-net)
 '((M1 (LOADINGS 2) (DRIVES 10 A A)))


(netlist-properties ex-net '(delays) F)

(netlist-properties ex-net '(delays) T)
 '((M1 (DELAYS ((70 235) (57 198) A) A A)))



(setq ex-net
      '((m1 (a) (b c d)
            ((o1 (b) b-not (a))
             (o2 (c) id (a))
             (o3 (d) id (b)))
            nil)))
```

```
(netlist-loadings-and-drives ex-net)
 '((M1 (LOADINGS 2) (DRIVES 10 A B)))


(netlist-properties ex-net '(delays) F)

(netlist-properties ex-net '(delays) T)
 '((M1 (DELAYS ((70 235) (57 198) A) A B)))



(setq ex-net
      '((m1 (a) (b c d)
            ((o1 (b) b-not (a))
             (o2 (c) id (b))
             (o3 (d) id (c)))
            nil)))


(netlist-loadings-and-drives ex-net)
 '((M1 (LOADINGS 2) (DRIVES 10 B B)))


(netlist-properties ex-net '(delays) F)

(netlist-properties ex-net '(delays) T)
 '((M1 (DELAYS ((70 235) (57 198) A) B B)))



(setq ex-net
      '((m1 (a) (c d)
            ((o1 (b) b-not (a))
             (o2 (c) id (b))
             (o3 (d) id (c)))
            nil)))


(netlist-loadings-and-drives ex-net)
 '((M1 (LOADINGS 2) (DRIVES 10 C)))


(netlist-properties ex-net '(delays) F)

(netlist-properties ex-net '(delays) T)
 '((M1 (DELAYS ((70 235) (57 198) A) C)))
```

```
(setq ex-net '((m3 (in0 clk) (out1 out2)
                   ((o3 (is3 is4) fd1 (is1 is2))
                    (o1 (is1) id (in0))
                    (o2 (is2) id (clk))
                    (o4 (out1) id (is3))
                    (o5 (out2) id (is4)))
                   o3)))


(netlist-loadings-and-drives ex-net)
 '((M3 (LOADINGS 1 1) (DRIVES 10 10)))


(netlist-properties ex-net '(delays) F)

(netlist-properties ex-net '(delays) T)
 '((M3 (DELAYS ((147 1024) (55 1288))
               ((145 1432) (53 1447)))))

|#
#|

(setq
 ex-net
 '((m1 (i1) (o1)
       ((super-big-long-name-for-an-occurrence-it-has-58-characters
         (o1) m2 (i1)))
       nil)
   (m2 (i1) (o1)
       ((another-super-big-long-name-for-an-occurrence-it-has-66-characters
         (o1) m3 (i1)))
       nil)
   (m3
    (i1) (o1)
    ((still-another-super-big-long-name-for-an-occurrence-it-has-72-characters
      (o1) m4 (i1)))
    nil)
   (m4 (i1) (super-big-long-name-for-a-signal-it-has-53-characters)
       ((yet-another-super-big-long-name-for-an-occurrence-it-has-70-characters
         (super-big-long-name-for-a-signal-it-has-53-characters) b-not (i1)))
       nil)))

; max-hierarchical-name-length: 324
```

```
(lsi-netlist-syntax-ok ex-net 64 255)
 (NET-ERROR 'LSI-NETLIST-SYNTAX-ERRORS
  (LIST
   (NET-ERROR
    '(MODULE M2)
    (LIST
     (NET-ERROR 'MODULE-OCCURRENCES
      (LIST
       (NET-ERROR
       '(OCCURRENCE
           ANOTHER-SUPER-BIG-LONG-NAME-FOR-AN-OCCURRENCE-IT-HAS-66-CHARACTERS)
       (LIST
        (NET-ERROR 'OCCURRENCE-NAME
         (LIST
          (NET-ERROR
'(
BAD-LSI-NAME ANOTHER-SUPER-BIG-LONG-NAME-FOR-AN-OCCURRENCE-IT-HAS-66-CHARACTERS
)
          (LIST (NET-ERROR 'NAME-TOO-LONG
                           '((NAME-LENGTH 66)
                             (MAX-ALLOWED-LENGTH 64)))))))
        (NET-ERROR 'HIERARCHICAL-NAME-TOO-LONG
 '((NAME-LENGTH 265)
   (MAX-ALLOWED-LENGTH 255)
   (NAME
    (ANOTHER-SUPER-BIG-LONG-NAME-FOR-AN-OCCURRENCE-IT-HAS-66-CHARACTERS
     STILL-ANOTHER-SUPER-BIG-LONG-NAME-FOR-AN-OCCURRENCE-IT-HAS-72-CHARACTERS
     YET-ANOTHER-SUPER-BIG-LONG-NAME-FOR-AN-OCCURRENCE-IT-HAS-70-CHARACTERS
     SUPER-BIG-LONG-NAME-FOR-A-SIGNAL-IT-HAS-53-CHARACTERS)))))))))
   (NET-ERROR
    '(MODULE M3)
    (LIST
     (NET-ERROR 'MODULE-OCCURRENCES
      (LIST
       (NET-ERROR
       '(OCCURRENCE
 STILL-ANOTHER-SUPER-BIG-LONG-NAME-FOR-AN-OCCURRENCE-IT-HAS-72-CHARACTERS)
       (LIST
        (NET-ERROR 'OCCURRENCE-NAME
         (LIST
          (NET-ERROR
 '(BAD-LSI-NAME
   STILL-ANOTHER-SUPER-BIG-LONG-NAME-FOR-AN-OCCURRENCE-IT-HAS-72-CHARACTERS)
          (LIST (NET-ERROR 'NAME-TOO-LONG
                           '((NAME-LENGTH 72)
                             (MAX-ALLOWED-LENGTH 64))))))))))))
   (NET-ERROR
    '(MODULE M4)
    (LIST
     (NET-ERROR 'MODULE-OCCURRENCES
      (LIST
       (NET-ERROR
       '(OCCURRENCE
  YET-ANOTHER-SUPER-BIG-LONG-NAME-FOR-AN-OCCURRENCE-IT-HAS-70-CHARACTERS)
```

```
       (LIST
        (NET-ERROR 'OCCURRENCE-NAME
         (LIST
          (NET-ERROR
          '(BAD-LSI-NAME
 YET-ANOTHER-SUPER-BIG-LONG-NAME-FOR-AN-OCCURRENCE-IT-HAS-70-CHARACTERS)
          (LIST (NET-ERROR 'NAME-TOO-LONG
                           '((NAME-LENGTH 70)
                             (MAX-ALLOWED-LENGTH 64)))))))))))))))

|#

#|

(setq net '((m1 (a b) (c)
                ((o1 (is1) t-wire (a b))
                 (o2 (is2) pullup (is1))
                 (o3  (c)  b-not  (is2)))
                nil)
            (m2 (a b clk) (c)
                ((o1 (is1) t-wire (a b))
                 (o2 (is2) pullup (is1))
                 (o3 (c cn) fd1 (is2 clk)))
                o3)))


(top-level-predicate net)
 T
```

```
(netlist-database net F)
 '((M1 (DELAYS ((70 235) (57 198) (OR A B)))
       (DRIVES 10)
       (INPUT-TYPES TRI-STATE TRI-STATE)
       (INPUTS A B)
       (LOADINGS 4 4)
       (OUT-DEPENDS (A B))
       (OUTPUT-TYPES BOOLP)
       (OUTPUTS C)
       (GATES . 1)
       (PADS . 0)
       (PRIMITIVES . 1)
       (TRANSISTORS . 3))
   (M2 (DELAYS ((147 1024) (55 1288)))
       (DRIVES 10)
       (INPUT-TYPES TRI-STATE TRI-STATE CLK)
       (INPUTS A B CLK)
       (LOADINGS 3 3 1)
       (OUT-DEPENDS NIL)
       (OUTPUT-TYPES BOOLP)
       (OUTPUTS C)
       (STATE-TYPES . BOOLP)
       (STATES . 03)
       (GATES . 7)
       (PADS . 0)
       (PRIMITIVES . 1)
       (TRANSISTORS . 26)))


(primitive-count 0 'b-and     'transistors nil)
 6

(primitive-count 0 'ttl-input 'pads nil)
 1

(primitive-count 0 'm2        'gates       net)
 7

(primitive-count 0 'm3        'pads        net)
 (NET-ERROR 'UNKNOWN-MODULE 'M3)

(let ((occs (module-occurrences (lookup-module 'm2 net))))
  (primitive-count 1 occs     'primitives  net))
 1

(primitive-count 2 'm3 'pads net)
 ; unknown flag



(out-depends 0 'fd1 '((d) (clk)) nil nil)
 '(NIL NIL)
```

```
(out-depends 0 'm1  '((x) (y))   nil net)
 '((X Y))

(let ((occs (module-occurrences (lookup-module 'm2 net))))
  (out-depends 1 occs  '(a b clk) '(c) net))
 (LIST '(IS2 CLK IS1 A B C)
       '(CN)
       T
       '(C)
       T
       '(IS2 A B)
       T
       '(IS1 A B)
       T
       '(A A)
       '(B B)
       '(CLK CLK))

(out-depends 2 'm1 '((x) (y)) nil net)
 ; unknown flag

(output-dependencies 'b-not '((x) (y)) nil)
 ; wrong number of args

(output-dependencies 'm3 '((x) (y)) net)
 ; unknown module m3

(output-dependencies 'm2 '((x) (y)) net)
 ; wrong number of args

(output-dependencies 'm2 '((x) (y) (clock)) net)
 '(NIL)

(output-dependencies 'm1 '((x) (y)) net)
 '((X Y))



(IO-types 0 'b-not '(free) nil nil)
 '((INPUT-TYPES BOOLP)
   (OUTPUT-TYPES BOOLP))

(IO-types 0 'id '(ttl) nil nil)
 '((INPUT-TYPES FREE)
   (OUTPUT-TYPES TTL))

(IO-types 0 'm1 '(tri-state tri-state) nil net)
 '((INPUT-TYPES TRI-STATE TRI-STATE)
   (OUTPUT-TYPES BOOLP))
```

```
(let ((occs (module-occurrences (lookup-module 'm2 net))))
  (IO-types 1 occs '(a b clk) '(c) net))
 '((INPUT-TYPES (CLK . CLK)
                (IS2 . BOOLP)
                (IS1 . TRI-STATE)
                (B . TRI-STATE)
                (A . TRI-STATE))
   (OUTPUT-TYPES (CN . BOOLP)
                 (C . BOOLP)
                 (IS2 . BOOLP)
                 (IS1 . TRI-STATE)))

(let ((occs (module-occurrences (lookup-module 'm2 net))))
  (IO-types 3 occs '(a b clk) '(c) net))
 ; unknown flag



(state-type-requirement 0 'ttl-bidirect nil)
 NIL

(state-type-requirement 0 'fd1 nil)
 'BOOLP

(let ((occs (module-occurrences (lookup-module 'm2 net))))
  (state-type-requirement 1 occs net))
 '((O3 . BOOLP))

(state-type-requirement 4 'm1 net)
 ; unknown flag



(loadings-and-drives 0 'b-not '(x) '(y) nil)
 '((LOADINGS 2) (DRIVES 10))

(loadings-and-drives 0 'm2 '(x y clock) '(out0) net)
 '((LOADINGS 3 3 1) (DRIVES 10))

(loadings-and-drives 0 'm3 '(x y clock) '(out0) net)
 ; unknown module m3

(let ((occs (module-occurrences (lookup-module 'm2 net))))
  (loadings-and-drives 1 occs '(a b clk) '(c) net))
 '((LOADINGS (B . 3)
             (A . 3)
             (IS1 . 2)
             (CLK . 1)
             (IS2 . 1))
   (DRIVES (CN . 10)
           (C . 10)
           (IS2 . IS1)
           (IS1 MIN A B)))
```

```
(loadings-and-drives 2 'm1 '(a b) '(c) net)
 ; unknown flag

|#
```

14.35 "primitives.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    PRIMITIVES.EVENTS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;   We create and prove DUAL-EVAL lemmas for all of the primitives.  These
;;;   lemmas are kept disabled since each is stored under the function symbol
;;;   DUAL-EVAL.   RESULT-AND-STATE-LEMMAS is defined in "primitives.lisp".


(result-and-state-lemmas)

;;;   Below are definitions of some simple circuits which we think of as
;;;   primitives, even though they are constructed from more than 1 macrocell.

;;;    B-BUF


(defn b-buf* ()
  '(b-buf (in) (out)
          ((g0 (out- out) b-nbuf (in)))
          nil))


(module-predicate b-buf*)


(module-netlist b-buf*)


(prove-lemma b-buf$value (rewrite)
  (implies
   (b-buf& netlist)
   (equal (dual-eval 0 'b-buf (list in) state netlist)
          (list (f-buf in))))
  ;;hint
  ((enable b-buf& boolp-b-gates b-buf boolp b-nbuf$value)))


(disable b-buf$value)

;;;   B-BUF-PWR -- In the LSI Logic implementation, this "super-buffer" can
;;;   drive 64 loads in 1.6ns.
```

```
(defn b-buf-pwr* ()
  '(b-buf-pwr (in) (out)
              ((g0 (out-) b-not (in))
               (g1 (out)  b-not-b4ip (out-)))
              nil))


(module-predicate b-buf-pwr*)


(module-netlist b-buf-pwr*)


(prove-lemma b-buf-pwr$value (rewrite)
  (implies
   (b-buf-pwr& netlist)
   (equal (dual-eval 0 'b-buf-pwr (list in) state netlist)
          (list (f-buf in))))
  ;;hint
  ((enable b-buf-pwr& boolp-b-gates b-buf b-not$value b-not-b4ip$value)))


(disable b-buf-pwr$value)



;;;   DP-RAM-16x32
;;;
;;;   We prefer to think of the DP-RAM-16x32 arguments as being
;;;   structured as two addresses, the write enable, and
;;;   the data.


(prove-lemma dp-ram-16x32-args-crock (rewrite)
  (implies
   (and (equal (length a) 4) (properp a)
        (equal (length b) 4) (properp b)
        (equal (length c) 32) (properp c))
   (equal (append a (append b (cons x c)))
          (append (list-as-collected-nth a 4 0)
                  (append (list-as-collected-nth b 4 0)
                          (cons x
                                (list-as-collected-nth c 32 0))))))
  ;;Hint
  ((use (equal-length-4-as-collected-nth (l a))
        (equal-length-4-as-collected-nth (l b))
        (equal-length-32-as-collected-nth (l c)))
   (disable open-list-as-collected-nth)))


(disable dp-ram-16x32-args-crock)
```

```
(prove-lemma dp-ram-16x32$structured-value (rewrite)
  (implies
   (and (dp-ram-16x32& netlist)
        (equal (length a) 4) (properp a)
        (equal (length b) 4) (properp b)
        (equal (length c) 32) (properp c))
   (equal (dual-eval 0 'dp-ram-16x32 (append a (append b (cons x c)))
                     state netlist)
          (dual-port-ram-value 32 4 (append a (append b (cons x c)))
                               state)))
   ;;Hint
   ((enable dp-ram-16x32-args-crock dp-ram-16x32$value)
    (disable car-cdr-elim dual-port-ram-value)))


(disable dp-ram-16x32$structured-value)


(prove-lemma dp-ram-16x32$structured-state (rewrite)
  (implies
   (and (dp-ram-16x32& netlist)
        (equal (length a) 4) (properp a)
        (equal (length b) 4) (properp b)
        (equal (length c) 32) (properp c))
   (equal (dual-eval 2 'dp-ram-16x32 (append a (append b (cons x c)))
                     state netlist)
          (dual-port-ram-state 32 4 (append a (append b (cons x c)))
                               state)))
   ;;Hint
   ((enable dp-ram-16x32-args-crock dp-ram-16x32$state)
    (disable car-cdr-elim dual-port-ram-state)))


(disable dp-ram-16x32$structured-state)




;;;   MEM-32x32
;;;
;;;   We prefer to think of the MEM-32x32 arguments as being
;;;   structured as an address, data-bus, a strobe, and a read-write
;;;   signal.
```

```
(prove-lemma mem-32x32-args-crock (rewrite)
  (implies
   (and (equal (length addr) 32)  (properp addr)
        (equal (length data) 32)  (properp data))
   (equal (cons rw-
                (cons strobe-
                      (append addr data)))
          (cons rw-
                (cons strobe-
                      (append (list-as-collected-nth addr 32 0)
                              (list-as-collected-nth data 32 0)))))))
  ;;Hint
  ((use (equal-length-32-as-collected-nth (l addr))
        (equal-length-32-as-collected-nth (l data)))
   (disable open-list-as-collected-nth)))


(disable mem-32x32-args-crock)
```

```
(prove-lemma list-32-nth-collapse (rewrite)
  (implies (and (properp args)
                (equal (length args) 32))
           (equal (list (nth 0 args)
                        (nth 1 args)
                        (nth 2 args)
                        (nth 3 args)
                        (nth 4 args)
                        (nth 5 args)
                        (nth 6 args)
                        (nth 7 args)
                        (nth 8 args)
                        (nth 9 args)
                        (nth 10 args)
                        (nth 11 args)
                        (nth 12 args)
                        (nth 13 args)
                        (nth 14 args)
                        (nth 15 args)
                        (nth 16 args)
                        (nth 17 args)
                        (nth 18 args)
                        (nth 19 args)
                        (nth 20 args)
                        (nth 21 args)
                        (nth 22 args)
                        (nth 23 args)
                        (nth 24 args)
                        (nth 25 args)
                        (nth 26 args)
                        (nth 27 args)
                        (nth 28 args)
                        (nth 29 args)
                        (nth 30 args)
                        (nth 31 args))
                  args))
  ((use (equal-length-32-as-collected-nth (l args)))))


(disable list-32-nth-collapse)
```

```
(prove-lemma mem-32x32$structured-value-1 (rewrite)
  (implies
   (and (mem-32x32& netlist)
        (equal (length addr) 32)   (properp addr)
        (equal (length data) 32)   (properp data))
   (equal (dual-eval 0 'mem-32x32
                     (cons rw-
                           (cons strobe-
                                 (append addr data)))
                     state netlist)
          (memory-value state strobe- rw-
                        (list (nth 0 addr)
                              (nth 1 addr)
                              (nth 2 addr)
                              (nth 3 addr)
                              (nth 4 addr)
                              (nth 5 addr)
                              (nth 6 addr)
                              (nth 7 addr)
                              (nth 8 addr)
                              (nth 9 addr)
                              (nth 10 addr)
                              (nth 11 addr)
                              (nth 12 addr)
                              (nth 13 addr)
                              (nth 14 addr)
                              (nth 15 addr)
                              (nth 16 addr)
                              (nth 17 addr)
                              (nth 18 addr)
                              (nth 19 addr)
                              (nth 20 addr)
                              (nth 21 addr)
                              (nth 22 addr)
                              (nth 23 addr)
                              (nth 24 addr)
                              (nth 25 addr)
                              (nth 26 addr)
                              (nth 27 addr)
                              (nth 28 addr)
                              (nth 29 addr)
                              (nth 30 addr)
                              (nth 31 addr))
                        (list (nth 0 data)
                              (nth 1 data)
                              (nth 2 data)
                              (nth 3 data)
                              (nth 4 data)
                              (nth 5 data)
                              (nth 6 data)
                              (nth 7 data)
                              (nth 8 data)
                              (nth 9 data)
                              (nth 10 data)
```

```
                              (nth 11 data)
                              (nth 12 data)
                              (nth 13 data)
                              (nth 14 data)
                              (nth 15 data)
                              (nth 16 data)
                              (nth 17 data)
                              (nth 18 data)
                              (nth 19 data)
                              (nth 20 data)
                              (nth 21 data)
                              (nth 22 data)
                              (nth 23 data)
                              (nth 24 data)
                              (nth 25 data)
                              (nth 26 data)
                              (nth 27 data)
                              (nth 28 data)
                              (nth 29 data)
                              (nth 30 data)
                              (nth 31 data)))))
  ;;Hint
  ((enable mem-32x32$value
           mem-32x32-args-crock
           mem-value
           open-subrange)
   (disable car-cdr-elim memory-value)))


(disable mem-32x32$structured-value-1)


(prove-lemma mem-32x32$structured-value (rewrite)
  (implies
   (and (mem-32x32& netlist)
        (equal (length addr) 32)   (properp addr)
        (equal (length data) 32)   (properp data))
   (equal (dual-eval 0 'mem-32x32
                     (cons rw-
                           (cons strobe-
                                 (append addr data)))
                     state netlist)
          (memory-value state strobe- rw- addr data)))
  ;;Hint
  ((enable mem-value list-32-nth-collapse
           mem-32x32$structured-value-1)
   (disable car-cdr-elim memory-value)))


(disable mem-32x32$structured-value)
```

```
(prove-lemma mem-32x32$structured-state-1 (rewrite)
  (implies
   (and (mem-32x32& netlist)
        (equal (length addr) 32)   (properp addr)
        (equal (length data) 32)   (properp data))
   (equal (dual-eval 2 'mem-32x32
                     (cons rw-
                           (cons strobe-
                                 (append addr data)))
                     state netlist)
          (next-memory-state state strobe- rw-
                             (list (nth 0 addr)
                                   (nth 1 addr)
                                   (nth 2 addr)
                                   (nth 3 addr)
                                   (nth 4 addr)
                                   (nth 5 addr)
                                   (nth 6 addr)
                                   (nth 7 addr)
                                   (nth 8 addr)
                                   (nth 9 addr)
                                   (nth 10 addr)
                                   (nth 11 addr)
                                   (nth 12 addr)
                                   (nth 13 addr)
                                   (nth 14 addr)
                                   (nth 15 addr)
                                   (nth 16 addr)
                                   (nth 17 addr)
                                   (nth 18 addr)
                                   (nth 19 addr)
                                   (nth 20 addr)
                                   (nth 21 addr)
                                   (nth 22 addr)
                                   (nth 23 addr)
                                   (nth 24 addr)
                                   (nth 25 addr)
                                   (nth 26 addr)
                                   (nth 27 addr)
                                   (nth 28 addr)
                                   (nth 29 addr)
                                   (nth 30 addr)
                                   (nth 31 addr))
                             (list (nth 0 data)
                                   (nth 1 data)
                                   (nth 2 data)
                                   (nth 3 data)
                                   (nth 4 data)
                                   (nth 5 data)
                                   (nth 6 data)
                                   (nth 7 data)
                                   (nth 8 data)
                                   (nth 9 data)
                                   (nth 10 data)
```

```
                                    (nth 11 data)
                                    (nth 12 data)
                                    (nth 13 data)
                                    (nth 14 data)
                                    (nth 15 data)
                                    (nth 16 data)
                                    (nth 17 data)
                                    (nth 18 data)
                                    (nth 19 data)
                                    (nth 20 data)
                                    (nth 21 data)
                                    (nth 22 data)
                                    (nth 23 data)
                                    (nth 24 data)
                                    (nth 25 data)
                                    (nth 26 data)
                                    (nth 27 data)
                                    (nth 28 data)
                                    (nth 29 data)
                                    (nth 30 data)
                                    (nth 31 data)))))
  ;;Hint
  ((enable mem-32x32$state
           mem-32x32-args-crock
           mem-state
           open-subrange)
   (disable car-cdr-elim next-memory-state)))


(disable mem-32x32$structured-state-1)


(prove-lemma mem-32x32$structured-state (rewrite)
  (implies
   (and (mem-32x32& netlist)
        (equal (length addr) 32)   (properp addr)
        (equal (length data) 32)   (properp data))
   (equal (dual-eval 2 'mem-32x32
                     (cons rw-
                           (cons strobe-
                                 (append addr data)))
                     state netlist)
          (next-memory-state state strobe- rw- addr data)))
  ;;Hint
  ((enable mem-state list-32-nth-collapse
           mem-32x32$structured-state-1)
   (disable car-cdr-elim memory-value)))


(disable mem-32x32$structured-state)
```

14.36 "unbound.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;   UNBOUND.EVENTS
;;;
;;;   For doing proofs of recursive module generators, it is often necessary to
;;;   know that a signal name is "unbound", i.e., does not appear further in
;;;   the generated body.  By a "body" we mean an occurrence body, and by
;;;   "bound" we mean assigned a value in the ALIST created by DUAL-EVAL with
;;;   flag 1.  This concept is eloquently summed up in the final two lemmas of
;;;   this file.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  UNBOUND-IN-BODY


(defn unbound-in-body (name body)
  (if (nlistp body)
      t
    (let ((occurrence (car body)))
      (let ((outputs (occ-outputs occurrence)))
        (and (not (member name outputs))
             (unbound-in-body name (cdr body)))))))


(disable unbound-in-body)


(prove-lemma unbound-in-body-nlistp (rewrite)
  (implies
   (nlistp body)
   (unbound-in-body name body))
  ;;Hint
  ((enable unbound-in-body)))


(prove-lemma unbound-in-body-listp (rewrite)
  (equal (unbound-in-body name (cons occurrence rest))
         (let ((outputs (occ-outputs occurrence)))
           (and (not (member name outputs))
                (unbound-in-body name rest))))
  ;;Hint
  ((enable unbound-in-body)))

;;;   ALL-UNBOUND-IN-BODY
```

```
(defn all-unbound-in-body (names body)
  (if (nlistp body)
      t
    (let ((occurrence (car body)))
      (let ((outputs (occ-outputs occurrence)))
        (and (disjoint names outputs)
             (all-unbound-in-body names (cdr body)))))))


(disable all-unbound-in-body)


(prove-lemma all-unbound-in-body-nlistp (rewrite)
  (implies
   (nlistp body)
   (all-unbound-in-body names body))
  ;;Hint
  ((enable all-unbound-in-body)))


(prove-lemma all-unbound-in-body-listp (rewrite)
  (equal (all-unbound-in-body names (cons occurrence rest))
         (let ((outputs (occ-outputs occurrence)))
           (and (disjoint names outputs)
                (all-unbound-in-body names rest))))
  ;;Hint
  ((enable all-unbound-in-body)))


(prove-lemma all-unbound-in-body-append (rewrite)
  (equal (all-unbound-in-body (append names1 names2) body)
         (and (all-unbound-in-body names1 body)
              (all-unbound-in-body names2 body)))
  ;;Hint
  ((enable all-unbound-in-body append)))


(prove-lemma all-unbound-in-body-cons (rewrite)
  (equal (all-unbound-in-body (cons name names) body)
         (and (unbound-in-body name body)
              (all-unbound-in-body names body)))
  ;;Hint
  ((enable all-unbound-in-body unbound-in-body append)))


(prove-lemma all-unbound-in-body-nlistp-names (rewrite)
  (implies
   (nlistp names)
   (all-unbound-in-body names body))
  ;;Hint
  ((enable all-unbound-in-body)))

;;;   Lemmas
```

```
(prove-lemma unbound-in-body-dual-eval-1 (rewrite)
  (implies
   (and (equal flag 1)
        (unbound-in-body name body))
   (equal (value name (dual-eval flag body bindings state-bindings netlist))
          (value name bindings)))
  ;;Hint
  ((induct (dual-eval flag body bindings state-bindings netlist))
   (enable unbound-in-body)))


(prove-lemma all-unbound-in-body-dual-eval-1 (rewrite)
  (implies
   (and (equal flag 1)
        (all-unbound-in-body names body))
   (equal (collect-value
           names
           (dual-eval flag body bindings state-bindings netlist))
          (collect-value names bindings)))
  ;;Hint
  ((induct (dual-eval flag body bindings state-bindings netlist))
   (enable all-unbound-in-body)))
```

14.37 "vector-module.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    VECTOR-MODULE.EVENTS
;;;
;;;  Automatic definition and proofs for simple linear vector modules of
;;;  primitives or other modules.  VECTOR-MODULE is defined in
;;;  "vector-macros.lisp".
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


;;;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    VECTOR-MODULE-INDUCTION
;;;
;;;
;;;  The induction scheme for vector modules.
;;;
;;;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn vector-module-induction (body m n bindings state-bindings netlist)
  (if (zerop n)
      t
    (vector-module-induction
     (cdr body)
     (add1 m)
     (sub1 n)
     (dual-eval-body-bindings 1 body bindings state-bindings netlist)
     state-bindings
     netlist)))

;;;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    V-BUF
;;;    V-OR
;;;    V-XOR
;;;    V-PULLUP
;;;    V-WIRE
;;;
;;;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(vector-module v-buf (g (y) b-buf (a)) ((v-threefix a)) :enable (f-buf))


(vector-module v-or (g (y) b-or (a b)) ((fv-or a b)))
```

```
(vector-module v-xor (g (y) b-xor (a b)) ((fv-xor a b)))

(vector-module v-pullup (g (y) pullup (a)) ((v-pullup a)))

(vector-module v-wire (g (y) t-wire (a b)) ((v-wire a b)))
```

14.38 "translate.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    TRANSLATE.EVENTS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  LISP-NETLIST converts a netlist that may include INDEX shells to one where
;;;  the indexed names have been collapsed into literal atoms.  For example,
;;;  the indexed name (INDEX 'G 0) become 'G_0.


(defn rev-0 (x)
  (rev1 x 0))


(defn number-to-digit (number)
  (nth number
       (cdr (unpack 'a0123456789))))


(defn number-to-list1 (number)
  (if (zerop number)
      0
      (cons (number-to-digit (remainder number 10))
            (number-to-list1 (quotient number 10)))))


(defn number-to-list (number)
  (if (zerop number)
      (cons (cadr (unpack 'a0)) 0)
      (rev-0 (number-to-list1 number))))
```

```
(defn lisp-netlist (netlist)
  (if (indexp netlist)
      (pack (append (unpack (i-name netlist))
                    (cons (cadr (unpack 'a_))
                          (number-to-list (i-num netlist)))))
  (if (nlistp netlist)
      netlist
      (cons (lisp-netlist (car netlist))
            (lisp-netlist (cdr netlist))))))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    Below are some utilities to count the number of primitives in the
;;;  FM9001.  COLLECT-PRIMITIVES just collects the different primitives
;;;  for a particular module, X0, in NETLIST.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn collect-primitives (flg x0 netlist)
  (case flg

    (0 (let ((fn x0))
         (let ((module (lookup-module fn netlist)))
           (if (or (primp fn) (not module))
               (list fn)
               (flatten-list
                (collect-primitives
                 1 (module-occurrences module)
                 (delete-module fn netlist)))))))

    (1 (let ((body x0))
         (if (nlistp body)
             nil
             (cons (collect-primitives
                    0 (occ-function (car body)) netlist)
                   (collect-primitives 1 (cdr body) netlist)))))

    (otherwise f))

  ((ord-lessp (cons (add1 (count netlist)) (count x0)))))
```

```
(defn count-primitives (flg x0 x1 type netlist)
  (case flg

    (0 (let ((fn x0))
         (if (primp fn)
             (if (primp-lookup fn type)
                 (primp2 fn type)
               f)
           (let ((module (lookup-module fn netlist)))
             (if (listp module)
                 (let ((m-ins  (module-inputs      module))
                       (m-outs (module-outputs     module))
                       (m-occs (module-occurrences module)))
                   (count-primitives 1 m-occs 0 type
                                     (delete-module fn netlist)))
               f)))))

    (1 (let ((body x0)
             (sum x1))
         (if (nlistp body)
             sum
           (let ((occ (car body)))
             (let ((o-outs (occ-outputs  occ))
                   (o-fn   (occ-function occ))
                   (o-ins  (occ-inputs   occ)))
               (if (count-primitives 0 o-fn nil type netlist)
                   (count-primitives
                    1 (cdr body)
                    (plus sum
                          (count-primitives 0 o-fn nil type netlist))
                    type
                    netlist)
                 f))))))
    (otherwise f))

  ((ord-lessp (cons (add1 (count netlist)) (count x0)))))


(disable count-primitives)


#|

;;;  In R-LOOP, the entire FM9001, sans the I/O pads, can be generated
;;;  and its primitives can be counted.


(setq chip-module-net          (lisp-netlist (chip-module$netlist)))

(setq first-chip-module        (car chip-module-net))

(setq first-chip-module-name   (module-name first-chip-module))
```

```
(setq first-chip-module-inputs (module-inputs first-chip-module))


(collect-primitives 0 first-chip-module chip-module-net)


(count-primitives 0 first-chip-module-name first-chip-module-inputs
                  'primitives chip-module-net)

(count-primitives 0 first-chip-module-name first-chip-module-inputs
                  'gates chip-module-net)

(count-primitives 0 first-chip-module-name first-chip-module-inputs
                  'transistors chip-module-net)


;;; print-ndl-form-to-file is in "translate.lisp".


(print-ndl-form-to-file (cdr (assoc 'chip-module-net r-alist))
                        "chip-module.net")


;;;  For the entire chip, including pads, use the following forms.


(setq pads-net (lisp-netlist (chip$netlist)))


(print-ndl-form-to-file (cdr (assoc 'pads-net r-alist))
                        "chip.net")

|#
```

14.39 "examples.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    EXAMPLES.EVENTS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  FULL-ADDER example.


(defn half-adder* ()
  '(half-adder
    (a b)
    (sum carry)
    ((g0 (sum)   b-xor (a b))
     (g1 (carry) b-and (a b)))
    nil))


(module-netlist half-adder*)

(module-predicate half-adder*)


(prove-lemma half-adder$value (rewrite)
  (implies
   (half-adder& netlist)
   (equal (dual-eval 0 'half-adder (list a b) state netlist)
          (list (f-xor a b)
                (f-and a b))))
  ;;Hint
  ((enable half-adder& b-xor$value b-and$value)))


(disable half-adder$value)


(defn full-adder* ()
  '(full-adder
    (a b c)
    (sum carry)
    ((t0 (sum1 carry1) half-adder (a b))
     (t1 (sum carry2)  half-adder (sum1 c))
     (t2 (carry)       b-or       (carry1 carry2)))
    nil))


(module-netlist full-adder*)
```

```
(module-predicate full-adder*)


(defn f$full-adder (a b c)
  (list (f-xor (f-xor a b) c)
        (f-or (f-and a b)
              (f-and (f-xor a b) c))))


(prove-lemma full-adder$value (rewrite)
  (implies
   (full-adder& netlist)
   (equal (dual-eval 0 'FULL-ADDER (list a b c) state netlist)
          (f$full-adder a b c)))
  ;;Hint
  ((enable full-adder& half-adder$value b-or$value)))


(defn full-adder (a b c)
  (list (b-xor3 a b c)
        (b-or (b-and a (b-or b c))
              (b-and b c))))


(prove-lemma f$full-adder=full-adder (rewrite)
  (implies
   (and (boolp a) (boolp b) (boolp c))
   (equal (f$full-adder a b c)
          (full-adder a b c))))

;;;  M1/M2 example.


(defn m1* ()
  '(m1
    (clk en sel d q)
    (q)
    ((mux   (b)    b-if  (sel d q))
     (latch (a an) fd1   (b clk))
     (tbuf  (q)    t-buf (en a)))
    latch))


(module-netlist m1*)

(module-predicate m1*)
```

```
(prove-lemma m1$value (rewrite)
  (implies
   (m1& netlist)
   (equal (dual-eval 0 'M1 (list clk en sel d q) state netlist)
          (list (ft-buf en state))))
  ;;Hint
  ((enable m1& b-if$value fd1$value t-buf$value ft-buf f-buf)))


(disable m1$value)


(prove-lemma m1$state (rewrite)
  (implies
   (m1& netlist)
   (equal (dual-eval 2 'M1 (list clk en sel d q) state netlist)
          (f-if sel d q)))
  ;;Hint
  ((enable m1& b-if$value fd1$value fd1$state t-buf$value f-if ft-buf f-buf)))


(disable m1$state)


(defn m2* ()
  '(m2
    (clk en0 en1 sel0 sel1 d0 d1)
    (q)
    ((occ0 (q0) m1     (clk en0 sel0 d0 q))
     (occ1 (q1) m1     (clk en1 sel1 d1 q))
     (wire (q)  t-wire (q0 q1)))
    (occ0 occ1)))


(module-netlist m2*)

(module-predicate m2*)


(prove-lemma m2$value (rewrite)
  (implies
   (m2& netlist)
   (equal (dual-eval 0 'M2
                     (list clk en0 en1 sel0 sel1 d0 d1) state netlist)
          (list (ft-wire (ft-buf en0 (car state))
                         (ft-buf en1 (cadr state))))))
  ;;Hint
  ((enable m2& m1$value m1$state t-wire$value)))


(disable m2$value)
```

```
(prove-lemma m2$state (rewrite)
  (implies
   (m2& netlist)
   (equal (dual-eval 2 'M2
                     (list clk en0 en1 sel0 sel1 d0 d1) state netlist)
          (let ((q (ft-wire (ft-buf en0 (car state))
                            (ft-buf en1 (cadr state)))))
            (list (f-if sel0 d0 q) (f-if sel1 d1 q)))))
  ;;Hint
  ((enable m2& m1$value m1$state t-wire$value)))


(disable m2$state)
```

14.40 "example-v-add.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    ADDERS.EVENTS
;;;
;;;    Various non-recursive and recursive adders.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  BV-ADDER


(defn bv-adder (c a b)
  ;; c is a bit, a and b are bit-vectors of some length n; this
  ;; function returns a bit vector of length n+1
  (if (nlistp a)
      (list c)
    (cons (xor c (xor (car a) (car b)))
          (bv-adder (or (and (car a) (car b))
                        (and (car a) c)
                        (and (car b) c))
                    (cdr a)
                    (cdr b)))))

;;;  BV-ADDER-BODY


(defn bv-adder-body (m n)
  ;; m is the starting point for the ``current index,'' and n is
  ;; the number of occurrences that we want to build.
  (if (zerop n)
      nil
    (cons (list #i(g m)                 ;label
                (list #i(sum m)         ;outputs
                      #i(carry (add1 m)))
                'full-adder             ;device type
                (list #i(a m)           ;inputs
                      #i(b m)
                      #i(carry m)))
          (bv-adder-body (add1 m) (sub1 n)))))


(disable bv-adder-body)

;;;  BV-ADDER* -- The BV-ADDER* lemmas are necessary for technical reasons
;;;  (having to do with the heuristics for the direction of EQUALity
;;;  substitution) that arise in proofs using BV-ADDER-BODY-LEMMA.
```

```
(defn bv-adder* (n)
  ;; n-bit wide input vectors
  (list #i(bv-adder n)              ;(index bv-adder n), intuitively bv-adder_n
        (cons #i(carry 0)           ;primary inputs are
              (append #i(a 0 n)     ; (carry_0 a_0 a_1 ... a_n-1
                      #i(b 0 n)))   ;          b_0 b_1 ... b_n-1)
        (append #i(sum 0 n)         ;outputs are
                (list #i(carry n))) ; (sum_0 sum_1 ... sum_n-1 carry_n)
        (bv-adder-body 0 n)         ;occurrences
        nil))                       ;NIL state


(destructuring-lemma bv-adder*)

;;(module-predicate bv-adder*), but not for parametrized modules


(defn bv-adder& (netlist n)
  (and (equal (lookup-module #i(bv-adder n) netlist) (bv-adder* n))
       (full-adder& (delete-module #i(bv-adder n) netlist))))


(disable bv-adder&)

;;(module-netlist bv-adder*), but not for parametrized modules


(defn bv-adder$netlist (n)
  (cons (bv-adder* n)
        (full-adder$netlist)))


(disable bv-adder$netlist)
```

```
(defn bv-adder-body$induction (m n bindings state-bindings netlist)
  ;; adapted from dual-eval-body-bindings
  (if (zerop n)
      bindings
    (let ((occ-name #i(g m))
          (outputs (list #i(sum m) #i(carry (add1 m))))
          (fn 'full-adder)
          (inputs (list #i(a m) #i(b m)
                        #i(carry m))))
      (bv-adder-body$induction
       (add1 m)
       (sub1 n)
       (append (pairlist outputs
                         (dual-eval 0
                                    fn
                                    (collect-value inputs bindings)
                                    (value occ-name state-bindings)
                                    netlist))
               bindings)
       state-bindings
       netlist))))


(prove-lemma bv-adder$unbound-in-body-sum (rewrite)
             (implies
              (lessp l m)
              (unbound-in-body #i(sum l)
                               (bv-adder-body m n)))
             ;;Hint
             ((enable unbound-in-body bv-adder-body)))


(prove-lemma bv-adder$unbound-in-body-carry
             (rewrite)
             (implies (lessp l (add1 m))
                      (unbound-in-body (index 'carry l)
                                       (bv-adder-body m n)))
             ((enable unbound-in-body bv-adder-body)))


(disable bv-adder$unbound-in-body-sum)

(disable bv-adder$unbound-in-body-carry)
```

```
(prove-lemma bv-adder-body$value (rewrite)
  (implies
   (and (full-adder& netlist)
        (boolp (value #i(carry m) bindings))
        (bvp (collect-value #i(a m n) bindings))
        (bvp (collect-value #i(b m n) bindings)))
   (equal (collect-value
           (append #i(sum m n) (list #i(carry (plus n m)))) ;outputs
           (dual-eval 1 (bv-adder-body m n)
                      bindings state-bindings netlist))
          (bv-adder (value #i(carry m) bindings)
                    (collect-value #i(a m n) bindings)
                    (collect-value #i(b m n) bindings))))
  ((enable bv-adder-body full-adder$value make-list fv-if-rewrite
           bv-adder$unbound-in-body-sum bv-adder$unbound-in-body-carry)
   (induct
    (bv-adder-body$induction m n
                             bindings state-bindings netlist))))


(prove-lemma bv-adder-body-special-case$value (rewrite)
  (implies
   (and (full-adder& netlist)
        (boolp (value #i(carry 0) bindings))
        (bvp (collect-value #i(a 0 n) bindings))
        (bvp (collect-value #i(b 0 n) bindings)))
   (equal (collect-value
           (append #i(sum 0 n) (list #i(carry n))) ;outputs
           (dual-eval 1 (bv-adder-body 0 n)
                      bindings state-bindings netlist))
          (bv-adder (value #i(carry 0) bindings)
                    (collect-value #i(a 0 n) bindings)
                    (collect-value #i(b 0 n) bindings))))
  ((use (bv-adder-body$value (m 0)))))


(prove-lemma bv-adder$value (rewrite)
  (implies
   (and (bv-adder& netlist n)
        (boolp c)
        (bvp a)
        (bvp b)
        (equal (length a) n)
        (equal (length b) n))
   (equal
    (dual-eval 0 #i(bv-adder n) (cons c (append a b)) state netlist)
    (bv-adder c a b)))
  ;; Hint
  ((enable bv-adder& bv-adder*$destructure full-adder$value)
   (disable collect-value-append)
   ;;(expand (dual-eval 0 'bv-adder (cons c (append a b)) state netlist))
   ))
```

```
(disable bv-adder-body-special-case$value)

(disable bv-adder-body$value)

(disable bv-adder$value)
```

14.41 "pg-theory.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;   PG-THEORY.EVENTS
;;;
;;;   A SUM/CARRY/GENERATE/PROPAGATE theory of addition, and the tree-based
;;;   adder TV-ADDER.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  SUM


(defn v-sum (c a b)
  (if (nlistp a)
      nil
    (cons (b-xor c (b-xor (car a) (car b)))
          (v-sum (b-or3 (b-and (car a) (car b))
                        (b-and (car a) c)
                        (b-and (car b) c))
                 (cdr a)
                 (cdr b)))))


(disable v-sum)


(prove-lemma bvp-sum (rewrite)
   (bvp (v-sum c a b))
   ;;Hint
   ((enable bvp v-sum)))


(prove-lemma length-sum (rewrite)
  (equal (length (v-sum c a b))
         (length a))
  ;;Hint
  ((enable length v-sum)))


(prove-lemma v-sum-congruence (rewrite)
  (implies
   c
   (equal (equal (v-sum c a b)
                 (v-sum t a b))
          t))
  ;;Hint
  ((enable v-sum)))
```

```
(prove-lemma a-1+1=a (rewrite)
  (implies
   (and (bvp a) c)
   (equal (v-sum c (v-not (nat-to-v 0 (length a))) a)
          a))
  ;;Hint
  ((enable v-sum v-not nat-to-v length)))

;;;  V-CARRY


(defn v-carry (c a b)
  (if (nlistp a)
      (boolfix c)
    (let ((a-bit (car a))
          (b-bit (car b)))
      (let ((p (or a-bit b-bit))
            (g (and a-bit b-bit)))
        (v-carry (or g (and p c)) (cdr a) (cdr b))))))


(disable v-carry)

;;;  V-PROPAGATE


(defn v-propagate (a b)
  (if (nlistp a)
      t
    (let ((a-bit (car a))
          (b-bit (car b)))
      (and (or a-bit b-bit)
           (v-propagate (cdr a) (cdr b))))))


(disable v-propagate)

;;;  V-GENERATE


(defn v-generate (a b)
  (if (nlistp a)
      f
    (let ((a-bit (car a))
          (b-bit (car b)))
      (or (v-generate (cdr a) (cdr b))
          (and a-bit b-bit (v-propagate (cdr a) (cdr b)))))))
```

```
(disable v-generate)

;;;  T-CARRY
;;;
;;;   We keep this non-recursive definition disabled because we are usually
;;;   interested in the logical properties of T-CARRY with respect to V-CARRY,
;;;   V-PROPAGATE, and V-GENERATE.


(defn f$t-carry (c prop gen)
  (f-or (f-and c prop) gen))


(disable f$t-carry)


(defn t-carry (c prop gen)
  (b-not (ao6 c prop gen)))


(disable t-carry)


(prove-lemma f$t-carry=t-carry (rewrite)
  (implies
   (and (boolp c)
        (boolp prop)
        (boolp gen))
   (equal (f$t-carry c prop gen)
          (t-carry c prop gen)))
  ;;Hint
  ((enable f$t-carry t-carry)))


(prove-lemma t-carry-congruence (rewrite)
  (implies
   x
   (equal (equal (t-carry x p g)
                 (t-carry t p g))
          t))
  ;;Hint
  ((enable t-carry)))


(defn t-carry* ()
  '(t-carry (c prop gen) (z)
            ((g0 (z-) ao6 (c prop gen))
             (g1 (z) b-not (z-)))
            nil))


(module-predicate t-carry*)
```

```
(module-netlist t-carry*)


(prove-lemma t-carry$value (rewrite)
  (implies
   (t-carry& netlist)
   (equal (dual-eval 0 't-carry (list c prop gen) state netlist)
          (list (f$t-carry c prop gen))))
  ;;Hint
  ((enable t-carry& f$t-carry b-not$value ao6$value)))


(disable t-carry$value)

;;;   Lemmas


(prove-lemma v-append-propagate (rewrite)
  (implies
   (equal (length a) (length c))
   (equal (v-propagate (append a b) (append c d))
          (and (v-propagate a c)
               (v-propagate b d))))
  ;;Hint
  ((enable length v-propagate append)))


(prove-lemma v-propagate-append (rewrite)
  (implies
   (equal (length a) (length c))
   (equal (b-and (v-propagate a c)
                 (v-propagate b d))
          (v-propagate (append a b) (append c d)))))


(disable v-propagate-append)


(prove-lemma generate-append (rewrite)
 (implies
   (equal (length a) (length c))
   (equal (t-carry (v-generate a c) (v-propagate b d) (v-generate b d))
          (v-generate (append a b) (append c d))))
 ;;Hint
 ((enable length t-carry v-generate v-propagate append)))
```

```
(prove-lemma append-sum (rewrite)
  (implies
   (and (equal (length a) (length b))
        (equal c2 (v-carry c1 a b)))
   (equal (append (v-sum c1 a b) (v-sum c2 c d))
          (v-sum c1 (append a c) (append b d))))
  ;;Hint
  ((enable length v-carry append v-sum boolfix)))


(prove-lemma t-carry-p-g-carry (rewrite)
  (equal (t-carry c (v-propagate a b) (v-generate a b))
         (v-carry c a b))
  ;;Hint
  ((enable t-carry v-propagate v-generate v-carry boolfix)))

;;;   Show that V-ADDER-OUTPUT == V-SUM, and V-ADDER-CARRY-OUT = V-CARRY


(prove-lemma v-adder-output=v-sum (rewrite)
  (equal (v-adder-output c a b)
         (v-sum c a b))
  ;;Hint
  ((enable v-adder v-sum firstn length)))


(prove-lemma v-adder-carry-out=v-carry (rewrite)
   (equal (v-adder-carry-out c a b)
          (v-carry c a b))
   ;;Hint
   ((enable v-adder v-carry nth restn length)))

;;;  TV-ADDER is a propagate-generate adder specification. C is the carry-in, A
;;;  and B are bit-vector addends, and TREE specifies the carry-lookahead
;;;  structure of the adder.  TV-ADDER conceptually returns 3 results:  The CAR
;;;  is (V-PROPAGATE A B), the CADR is (V-GENERATE A B), and the CDDR is
;;;  (V-SUM C A B).
```

```
(defn tv-adder (c a b tree)
  (if (nlistp tree)
      (list (b-or (car a) (car b))
            (b-and (car a) (car b))
            (b-xor (car a) (b-xor (car b) c)))
    (let ((lhs (tv-adder c
                           (tfirstn a tree)
                           (tfirstn b tree)
                           (car tree))))
      (let ((lhs-p (car lhs))
            (lhs-g (cadr lhs))
            (lhs-sum (cddr lhs)))
        (let ((rhs-c (t-carry c lhs-p lhs-g)))
          (let ((rhs (tv-adder rhs-c
                                 (trestn a tree)
                                 (trestn b tree)
                                 (cdr tree))))
            (let ((rhs-p (car rhs))
                  (rhs-g (cadr rhs))
                  (rhs-sum (cddr rhs)))
              (let ((p (b-and lhs-p rhs-p))
                    (g (t-carry lhs-g rhs-p rhs-g)))
                (cons p (cons g (append lhs-sum rhs-sum))))))))))


(disable tv-adder)

;;;  The proofs that TV-ADDER = (P . (G . SUM)).


(prove-lemma car-tv-adder-as-p (rewrite)
  (implies
   (and (equal (length a) (tree-size tree))
        (equal (length b) (tree-size tree)))
   (equal (car (tv-adder c a b tree))
          (v-propagate a b)))
  ;;Hint
  ((induct (tv-adder c a b tree))
   (disable b-and v-append-propagate)
   (enable v-propagate-append v-propagate)
   (expand (tv-adder c a b tree))))
```

```
(prove-lemma cadr-tv-adder-as-g (rewrite)
  (implies
   (and (equal (length a) (tree-size tree))
        (equal (length b) (tree-size tree)))
   (equal (cadr (tv-adder c a b tree))
          (v-generate a b)))
  ;;Hint
  ((induct (tv-adder c a b tree))
   (disable b-and v-append-propagate t-carry-p-g-carry)
   (enable v-propagate v-generate v-propagate-append)
   (expand (tv-adder c a b tree))))


(prove-lemma cddr-tv-adder-as-sum (rewrite)
  (implies
   (and (equal (length a) (tree-size tree))
        (equal (length b) (tree-size tree)))
   (equal (cddr (tv-adder c a b tree))
          (v-sum c a b)))
  ;;Hint
  ((induct (tv-adder c a b tree))
   (disable b-and t-carry v-append-propagate)
   (enable v-sum v-propagate-append)
   (expand (tv-adder c a b tree))))


(prove-lemma equality-of-three-element-lists ()
  (implies
   (and (not (equal x 0))
        (not (equal y 0))
        (not (equal z 0)))
   (equal (equal w (cons x (cons y z)))
          (and (equal (car w) x)
               (equal (cadr w) y)
               (equal (cddr w) z)))))


(prove-lemma tv-adder-as-p-g-sum (rewrite)
  (implies
   (and (equal (length a) (tree-size tree))
        (equal (length b) (tree-size tree)))
   (equal (tv-adder c a b tree)
          (cons (v-propagate a b)
                (cons (v-generate a b)
                      (v-sum c a b)))))
  ;;Hint
  ((disable tv-adder boolp)
   (use (equality-of-three-element-lists
         (w (tv-adder c a b tree))
         (x (v-propagate a b))
         (y (v-generate a b))
         (z (v-sum c a b))))))
```

```
(disable car-tv-adder-as-p)

(disable cadr-tv-adder-as-g)

(disable cddr-tv-adder-as-sum)

(disable tv-adder-as-p-g-sum)
```

14.42 "tv-if.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    TV-IF.EVENTS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  TV-IF is a vector multiplexor which buffers the control line according
;;;  to the TREE argument.  Buffers are inserted whenever a tree has
;;;  ((TREE-HEIGHT tree) modulo 3) = 0.
;;;
;;;  This generator creates modules which are to be used as in this sample
;;;  module occurence, where n = (tree-size tree):
;;;
;;;  (LIST <occurence-name>
;;;        <output list (n elements)>
;;;        (INDEX 'TV-IF (TREE-NUMBER tree))
;;;        (CONS <control signal>
;;;              <A input bus (n elements)>
;;;              <B input bus (n elements)>))
;;;
;;;  The predicate is (TV-IF& tree), and the netlist is (TV-IF$NETLIST tree).
;;;
;;;  For a balanced tree of n leaves, use tree = (MAKE-TREE n).
```

```
(defn tv-if-body (tree)
  (let ((a-names (indices 'a 0 (tree-size tree)))
        (b-names (indices 'b 0 (tree-size tree)))
        (out-names (indices 'out 0 (tree-size tree))))
    (let ((left-a-names (tfirstn a-names tree))
          (right-a-names (trestn a-names tree))
          (left-b-names (tfirstn b-names tree))
          (right-b-names (trestn b-names tree))
          (left-out-names (tfirstn out-names tree))
          (right-out-names (trestn out-names tree)))
      (if (nlistp tree)
          (list
           (list 'leaf
                 (list (index 'out 0))
                 'b-if
                 (list 'c (index 'a 0) (index 'b 0))))
        ;;  The control line is heuristically buffered.
        (let ((buffer? (zerop (remainder (tree-height tree) 3))))
          (let ((c-name (if buffer? 'c-buf 'c)))
            (append
             ;;  The buffer.
             (if buffer?
                 '((c-buf (c-buf) b-buf (c)))
               nil)
             (list
              ;;  The LHS tree.
              (list 'left
                    left-out-names
                    (index 'tv-if (tree-number (car tree)))
                    (cons c-name (append left-a-names left-b-names)))
              ;;  The RHS tree.
              (list 'right
                    right-out-names
                    (index 'tv-if (tree-number (cdr tree)))
                    (cons c-name (append right-a-names right-b-names))))))))))))


(defn tv-if* (tree)
  (let ((a-names (indices 'a 0 (tree-size tree)))
        (b-names (indices 'b 0 (tree-size tree)))
        (out-names (indices 'out 0 (tree-size tree))))
    (list
     ;;  Name
     (index 'tv-if (tree-number tree))
     ;;  Inputs
     (cons 'c (append a-names b-names))
     ;;  Outputs
     out-names
     ;;  Occurrences
     (tv-if-body tree)
     ;;  States
     nil)))
```

```
(destructuring-lemma tv-if*)

;;;  Note that both the netlist generator and the netlist predicate are
;;;  recursive.


(defn tv-if& (netlist tree)
  (if (nlistp tree)
      (and (equal (lookup-module (index 'tv-if (tree-number tree)) netlist)
                  (tv-if* tree))
           (b-if& (delete-module (index 'tv-if (tree-number tree)) netlist)))
    (and (equal (lookup-module (index 'tv-if (tree-number tree)) netlist)
                (tv-if* tree))
         (b-buf& (delete-module (index 'tv-if (tree-number tree)) netlist))
         (tv-if& (delete-module (index 'tv-if (tree-number tree)) netlist)
                 (car tree))
         (tv-if& (delete-module (index 'tv-if (tree-number tree)) netlist)
                 (cdr tree)))))


(disable tv-if&)


(defn tv-if$netlist (tree)
  (if (nlistp tree)
      (cons (tv-if* tree) (b-if$netlist))
    (cons (tv-if* tree)
          (union (tv-if$netlist (car tree))
                 (union (tv-if$netlist (cdr tree))
                        (b-buf$netlist))))))

;;;  The proofs require this special induction scheme.
```

```
(defn tv-if-induction (tree c a b state netlist)
  (if (nlistp tree)
      t
    (and
     (tv-if-induction
      (car tree)
      c
      (tfirstn a tree)
      (tfirstn b tree)
      0
      (delete-module (index 'tv-if (tree-number tree)) netlist))
     (tv-if-induction
      (car tree)
      (x)
      (tfirstn a tree)
      (tfirstn b tree)
      0
      (delete-module (index 'tv-if (tree-number tree)) netlist))
     (tv-if-induction
      (cdr tree)
      c
      (trestn a tree)
      (trestn b tree)
      0
      (delete-module (index 'tv-if (tree-number tree)) netlist))
     (tv-if-induction
      (cdr tree)
      (x)
      (trestn a tree)
      (trestn b tree)
      0
      (delete-module (index 'tv-if (tree-number tree)) netlist)))))

;;;  This lemma is necessary to force consideration of the output vector as an
;;;  APPEND of two sublists, based on the tree specification.
;;;  Expressions such as this would normally be rewritten the other way.
```

```
(prove-lemma tv-if-lemma-crock (rewrite)
  (implies
   (tv-if& (delete-module (index 'tv-if (tree-number tree)) netlist)
           (car tree))
   (equal (collect-value (indices 'out 0 n)
                         alist)
          (append (collect-value (firstn (tree-size (car tree))
                                         (indices 'out 0 n))
                                 alist)
                  (collect-value (restn (tree-size (car tree))
                                        (indices 'out 0 n))
                                 alist))))
  ;;Hint
  ((use (collect-value-splitting-crock
         (l (indices 'out 0 n))
         (n (tree-size (car tree)))))))


(disable tv-if-lemma-crock)


(prove-lemma tv-if$value (rewrite)
  (implies
   (and (tv-if& netlist tree)
        (properp a) (properp b)
        (equal (length a) (tree-size tree))
        (equal (length b) (tree-size tree)))
   (equal (dual-eval 0 (index 'tv-if (tree-number tree))
                     (cons c (append a b)) state netlist)
          (fv-if c a b)))
  ;;Hint
  ((induct (tv-if-induction tree c a b state netlist))
   (enable tv-if& tv-if-lemma-crock tv-if*$destructure
           b-buf$value b-if$value fv-if-rewrite
           tree-size)
   (disable open-indices)
   (expand (tv-if& netlist tree)
           (fv-if c a b))))


(disable tv-if$value)
```

14.43 "t-or-nor.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    T-OR-NOR.EVENTS -- A tree based, reducing OR-NOR.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn t-or-nor-body (tree parity)
  (let ((a-names (indices 'a 0 (tree-size tree))))
    (let ((left-a-names (tfirstn a-names tree))
          (right-a-names (trestn a-names tree)))
      (if (nlistp tree)
          (list (list 'leaf
                      (list 'out)
                      (if parity 'b-not 'b-buf)
                      (list (index 'a 0))))

        (if (and (nlistp (car tree))
                 (nlistp (cdr tree)))
            (list (list 'leaf
                        (list 'out)
                        (if parity 'b-nor 'b-or)
                        (list (index 'a 0) (index 'a 1))))
          (list
           ;;  The LHS tree.
           (list 'left
                 (list 'left-out)
                 (index (if parity 't-or 't-nor) (tree-number (car tree)))
                 left-a-names)
           ;;  The RHS tree.
           (list 'right
                 (list 'right-out)
                 (index (if parity 't-or 't-nor) (tree-number (cdr tree)))
                 right-a-names)
           (list 'output
                 (list 'out)
                 (if parity 'b-nor 'b-nand)
                 (list 'left-out 'right-out)))))))
```

```
(defn t-or-nor* (tree parity)
  (let ((a-names (indices 'a 0 (tree-size tree))))
    (list
     ;;  Name
     (index (if parity 't-nor 't-or) (tree-number tree))
     ;;  Inputs
     a-names
     ;;  Outputs
     (list 'out)
     ;;  Occurrences
     (t-or-nor-body tree parity)
     ;;  States
     nil)))


(destructuring-lemma t-or-nor*)


(defn t-or-nor& (netlist tree parity)
  (let ((delete-result (delete-module (index (if parity 't-nor 't-or)
                                             (tree-number tree))
                                      netlist))
        (lookup-okp (equal (lookup-module (index (if parity 't-nor 't-or)
                                                 (tree-number tree))
                                          netlist)
                           (t-or-nor* tree parity))))
    (let ((primitive& (and (b-not& delete-result)
                           (b-buf& delete-result)
                           (b-nor& delete-result)
                           (b-or& delete-result)
                           (b-nand& delete-result))))

      (if (or (nlistp tree)
              (and (nlistp (car tree))
                   (nlistp (cdr tree))))
          (and lookup-okp primitive&)

        (and lookup-okp
             (t-or-nor& delete-result (car tree) (not parity))
             (t-or-nor& delete-result (cdr tree) (not parity))
             primitive&)))))


(disable t-or-nor&)
```

```
(defn t-or-nor$netlist (tree parity)
  (if (or (nlistp tree)
          (and (nlistp (car tree))
               (nlistp (cdr tree))))
      (cons (t-or-nor* tree parity)
            (union (union (union (b-not$netlist) (b-buf$netlist))
                          (union (b-nor$netlist) (b-or$netlist)))
                   (b-nand$netlist)))
    (cons (t-or-nor* tree parity)
          (union (t-or-nor$netlist (car tree) (not parity))
                 (t-or-nor$netlist (cdr tree) (not parity))))))


(defn t-or-nor-induction (tree parity call-name a state netlist)
  (if (or (nlistp tree)
          (and (nlistp (car tree))
               (nlistp (cdr tree))))
      t
    (and (t-or-nor-induction (car tree) (not parity)
                             (if (not parity) 't-nor 't-or)
                             (tfirstn a tree) 0
                             (delete-module (index (if parity 't-nor 't-or)
                                                   (tree-number tree))
                                            netlist))
         (t-or-nor-induction (cdr tree) (not parity)
                             (if (not parity) 't-nor 't-or)
                             (trestn a tree) 0
                             (delete-module (index (if parity 't-nor 't-or)
                                                   (tree-number tree))
                                            netlist)))))


(defn tr-or-nor (a parity tree)
  (if (nlistp tree)
      (if parity (f-not (car a)) (f-buf (car a)))
    (if (and (nlistp (car tree))
             (nlistp (cdr tree)))
        (if parity
            (f-nor (car a) (cadr a))
          (f-or (car a) (cadr a)))
      (if parity
          (f-nor (tr-or-nor (tfirstn a tree) (not parity) (car tree))
                 (tr-or-nor (trestn a tree) (not parity) (cdr tree)))
        (f-nand (tr-or-nor (tfirstn a tree) (not parity) (car tree))
                (tr-or-nor (trestn a tree) (not parity) (cdr tree)))))))


(disable tr-or-nor)
```

```
(prove-lemma t-or-nor$value (rewrite)
  (implies
   (and (t-or-nor& netlist tree parity)
        (equal call-name (if parity 't-nor 't-or))
        (properp a)
        (equal (length a) (tree-size tree)))
   (equal (dual-eval 0 (index call-name (tree-number tree))
                     a state netlist)
          (list (tr-or-nor a parity tree))))
  ;;Hint
  ((induct (t-or-nor-induction tree parity call-name a state netlist))
   (enable t-or-nor*$destructure
           b-buf$value b-nand$value b-not$value b-or$value
           b-nor$value tr-or-nor)
   (disable tree-size-nlistp)
   (disable-theory f-gates)
   (expand (t-or-nor& netlist tree f)
           (t-or-nor& netlist tree parity)
           (tr-or-nor a f tree)
           (tr-or-nor a parity tree))))


(disable t-or-nor$value)


(defn btr-or-nor (a parity tree)
  (if (nlistp tree)
      (if parity (b-not (car a)) (b-buf (car a)))
    (if (and (nlistp (car tree))
             (nlistp (cdr tree)))
        (if parity
            (b-nor (car a) (cadr a))
          (b-or (car a) (cadr a)))
      (if parity
          (b-nor (btr-or-nor (tfirstn a tree) (not parity) (car tree))
                 (btr-or-nor (trestn a tree) (not parity) (cdr tree)))
        (b-nand (btr-or-nor (tfirstn a tree) (not parity) (car tree))
                (btr-or-nor (trestn a tree) (not parity) (cdr tree)))))))


(disable btr-or-nor)


(prove-lemma tr-or-nor=btr-or-nor (rewrite)
  (implies (and (bvp a)
                (equal (length a) (tree-size tree)))
           (equal (tr-or-nor a parity tree)
                  (btr-or-nor a parity tree)))
  ((disable tree-size-nlistp)
   (enable firstn restn
           expand-f-functions
           btr-or-nor tr-or-nor bvp)))
```

```
(prove-lemma btr-or-is-v-nzerop (rewrite)
  (implies
   (equal (length a) (tree-size tree))
   (equal (btr-or-nor a parity tree)
          (if parity
              (v-zerop a)
            (v-nzerop a))))
  ;;Hint
  ((induct (btr-or-nor a parity tree))
   (enable btr-or-nor v-nzerop tree-size)
   (expand (btr-or-nor a parity tree))))


;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;   TV-ZEROP* tree
;;;
;;;   A zero-detector module built from T-OR-NOR*.  The choice of
;;;   implementation is optimized for balanced trees.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn tv-zerop* (tree)
  (let ((in-names (indices 'in 0 (tree-size tree)))
        (odd-height (equal (remainder (tree-height tree) 2) 1)))
    (list
     ;; Name
     (index 'tv-zerop (tree-number tree))
     ;; Inputs
     in-names
     ;; Output
     '(out)
     ;; Body
     (if odd-height
         (list
          (list 'g0 '(out-) (index 't-or (tree-number tree)) in-names)
          (list 'g1 '(out)  'b-not                            '(out-)))
       (list
        (list 'g0 '(out) (index 't-nor (tree-number tree)) in-names)))
     ;; States
     nil)))


(destructuring-lemma tv-zerop*)
```

```
(defn tv-zerop& (netlist tree)
  (let ((in-names (indices 'in 0 (tree-size tree)))
        (odd-height (equal (remainder (tree-height tree) 2) 1)))
    (and (equal (lookup-module (index 'tv-zerop (tree-number tree)) netlist)
                (tv-zerop* tree))
         (let ((netlist
                (delete-module (index 'tv-zerop (tree-number tree)) netlist)))
           (and (t-or-nor& netlist tree (not odd-height))
                (b-not& netlist))))))


(disable tv-zerop&)


(defn tv-zerop$netlist (tree)
  (let ((odd-height (equal (remainder (tree-height tree) 2) 1)))
    (cons (tv-zerop* tree)
          (union (t-or-nor$netlist tree (not odd-height))
                 (b-not$netlist)))))



(defn f$tv-zerop (a tree)
  (let ((odd-height (equal (remainder (tree-height tree) 2) 1)))
    (if odd-height
        (f-not (tr-or-nor a f tree))
      (tr-or-nor a t tree))))


(disable f$tv-zerop)


(prove-lemma tv-zerop$value (rewrite)
  (implies
   (and (tv-zerop& netlist tree)
        (equal (length a) (tree-size tree))
        (properp a))
   (equal (dual-eval 0 (index 'tv-zerop (tree-number tree)) a state netlist)
          (list (f$tv-zerop a tree))))
  ;;Hint
  ((enable tv-zerop& tv-zerop*$destructure b-not$value t-or-nor$value
           f$tv-zerop)
   (disable-theory f-gates)))


(disable tv-zerop$value)
```

```
(prove-lemma f$tv-zerop=v-zerop (rewrite)
  (implies
   (and (equal (length a) (tree-size tree))
        (bvp a))
   (equal (f$tv-zerop a tree)
          (v-zerop a)))
  ;;Hint
  ((enable f$tv-zerop)
   (disable-theory f-gates)))
```

14.44 "fast-zero.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;  FAST-ZERO
;;;
;;;  A zero detector optimized for quick detection of the last 2 bits of the
;;;  input vector.  It should save a few nanoseconds in the FM9001.
;;;
;;;  LSI Logic timing analysis of the final design showed that this "fast"
;;;  zero-detector was about the same as simple, fully-balanced zero-detectors
;;;  defined in "t-or-nor.events".
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn f$fast-zero (v)
  (f-nor3 (tr-or-nor (firstn (sub1 (sub1 (length v))) v)
                     f
                     (make-tree (sub1 (sub1 (length v)))))
          (nth (sub1 (sub1 (length v))) v)
          (nth (sub1 (length v)) v)))


(disable f$fast-zero)


(prove-lemma f$fast-zero=tr-or-nor ()
  (implies
   (and (properp v)
        (geq (length v) 3))
   (equal (f$fast-zero v)
          (tr-or-nor v t (cons (make-tree (sub1 (sub1 (length v))))
                               (cons 0 0)))))
  ;;Hint
  ((enable f$fast-zero tr-or-nor f-nor3 f-nor nth-restn cdr-restn)
   (disable-theory f-gates)))


(prove-lemma f$fast-zero=v-zerop (rewrite)
  (implies
   (and (bvp v)
        (geq (length v) 3))
   (equal (f$fast-zero v)
          (v-zerop v)))
  ;;Hint
  ((use (f$fast-zero=tr-or-nor))
   (enable tree-size)))

;;;  Hardware
```

```
(module-generator
 (fast-zero* n)
 #i(fast-zero n)
 #i(a 0 n)
 '(z)
 (list
  (list 'front '(zfront) #i(t-or (tree-number (make-tree (sub1 (sub1 n)))))
        (firstn (sub1 (sub1 n)) #i(a 0 n)))
  (list 'result '(z) 'b-nor3
        (list 'zfront #i(a (sub1 (sub1 n))) #i(a (sub1 n)))))
 nil)


(defn fast-zero& (netlist n)
  (and (equal (lookup-module #i(fast-zero n) netlist) (fast-zero* n))
       (let ((netlist (delete-module #i(fast-zero n) netlist)))
         (and (t-or-nor& netlist (make-tree (sub1 (sub1 n))) f)
              (b-nor3& netlist)))))


(disable fast-zero&)


(defn fast-zero$netlist (n)
  (cons (fast-zero* n)
        (union (t-or-nor$netlist (make-tree (sub1 (sub1 n))) f)
               (b-nor3$netlist))))


(prove-lemma check-fast-zero$netlist ()
  (fast-zero& (fast-zero$netlist 5) 5))


(prove-lemma fast-zero$value (rewrite)
  (implies
   (and (fast-zero& netlist n)
        (properp v)
        (equal (length v) n)
        (geq n 3))
   (equal (dual-eval 0 #i(fast-zero n) v state netlist)
          (list (f$fast-zero v))))
  ;;Hint
  ((enable fast-zero& f$fast-zero fast-zero*$destructure t-or-nor$value
           b-nor3$value)
   (disable open-indices)
   (disable-theory f-gates)))
```

14.45 "v-equal.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    V-EQUAL.EVENTS
;;;
;;;  An n-bit equality circuit -- An XOR vector and a zero detector.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(module-generator
 (v-equal* n)
 (index 'V-EQUAL n)
 (append (indices 'A 0 n) (indices 'B 0 n))
 '(EQUAL)
 (list
  (list 'G0
        (indices 'X 0 n)
        (index 'V-XOR n)
        (append (indices 'A 0 n) (indices 'B 0 n)))
  (list 'G1
        '(EQUAL)
        (index 'TV-ZEROP (tree-number (make-tree n)))
        (indices 'X 0 n)))
 nil)


(defn v-equal& (netlist n)
  (and (equal (lookup-module (index 'V-EQUAL n) netlist)
              (v-equal* n))
       (let ((netlist (delete-module (index 'V-EQUAL n) netlist)))
         (and (v-xor& netlist n)
              (tv-zerop& netlist (make-tree n))))))


(disable v-equal&)


(defn v-equal$netlist (n)
  (cons (v-equal* n)
        (union (v-xor$netlist n)
               (tv-zerop$netlist (make-tree n)))))


(defn f$v-equal (a b)
  (f$tv-zerop (fv-xor a b) (make-tree (length a))))


(disable f$v-equal)
```

```
(prove-lemma v-equal$value (rewrite)
  (implies
   (and (v-equal& netlist n)
        (not (zerop n))
        (properp a) (properp b)
        (equal (length a) n)
        (equal (length b) n))
   (equal (dual-eval 0 (index 'V-EQUAL n) (append a b) state netlist)
          (list (f$v-equal a b))))
  ;;Hint
  ((enable v-equal& v-xor$value tv-zerop$value v-equal*$destructure
           f$v-equal)
   (disable open-indices)))


(disable v-equal$value)


(prove-lemma f$v-equal=equal* (rewrite)
  (implies
   (and (not (zerop (length a)))
        (bvp a)
        (bvp b)
        (equal (length a) (length b)))
   (equal (f$v-equal a b)
          (equal a b)))
  ;;Hint
  ((enable f$v-equal)))
```

14.46 "v-inc4.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;   V-INC4.EVENTS -- A 4-bit incrementer.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn v-inc4 (a0 a1 a2 a3)
  (let ((a0n (b-not a0))
        (a1n (b-not a1))
        (a2n (b-not a2))
        (a3n (b-not a3)))
    (let ((c0 a0n)
          (c1 (b-nor a0n a1n))
          (c2 (b-nor3 a0n a1n a2n)))
      (list a0n
            (b-xor a1n c0)
            (b-equv a2n c1)
            (b-equv a3n c2)))))


(defn-to-module v-inc4)


(defn f$v-inc4$v (a)
  (let ((a0 (car a))
        (a1 (cadr a))
        (a2 (caddr a))
        (a3 (cadddr a)))
    (let ((a0n (f-not a0))
          (a1n (f-not a1))
          (a2n (f-not a2))
          (a3n (f-not a3)))
      (let ((c0 a0n)
            (c1 (f-nor a0n a1n))
            (c2 (f-nor3 a0n a1n a2n)))
        (list a0n
              (f-xor a1n c0)
              (f-equv a2n c1)
              (f-equv a3n c2))))))


(disable f$v-inc4$v)
```

```
(prove-lemma properp-length-f$v-inc4$v (rewrite)
  (and (properp (f$v-inc4$v a))
       (equal (length (f$v-inc4$v a)) 4))
  ;;Hint
  ((enable f$v-inc4$v)
   (disable-theory f-gates)))


(prove-lemma v-inc4$value-as-v-inc (rewrite)
  (implies
   (and (v-inc4& netlist)
        (properp a)
        (equal (length a) 4))
   (equal (dual-eval 0 'v-inc4 a state netlist)
          (f$v-inc4$v a)))
  ;;Hint
  ((enable v-inc4$value f$v-inc4 f$v-inc4$v equal-length-add1)
   (disable-theory f-gates)))


(prove-lemma f$v-inc4$v=v-inc (rewrite)
  (implies
   (and (bvp a)
        (equal (length a) 4))
   (equal (f$v-inc4$v a)
          (v-inc a)))
  ;;Hint
  ((enable f$v-inc4$v v-inc v-sum equal-length-add1 boolp-b-gates)
   (disable-theory f-gates)))
```

14.47 "tv-dec-pass.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    TV-DEC-PASS.EVENTS
;;;
;;;   The output of the decrement/pass unit is either the input, or the input
;;;   decremented by 1.  This unit computes A - C, where A is the input vector,
;;;   and C is a 1-bit control signal representing 1 or 0.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  TV-DEC-PASS is a Boolean specification for a decrement/pass unit with a
;;;  carry-lookahead structure specified by the TREE input.  Carry-lookahead is
;;;  based on a single generate bit.  The CAR of the output vector is the
;;;  generate bit,  and the CDR of the output vector is A if C is high, else
;;;  A - 1.  Note that the parity of C is opposite from that specified above.


(defn tv-dec-pass (c a tree)
  (if (nlistp tree)
      (list (b-buf (car a)) (b-equv (car a) c))
    (let ((lhs (tv-dec-pass c (tfirstn a tree) (car tree))))
      (let ((rhs (tv-dec-pass (b-or c (car lhs)) (trestn a tree) (cdr tree))))
        (cons (b-or (car lhs) (car rhs))
              (append (cdr lhs) (cdr rhs)))))))


(disable tv-dec-pass)


(prove-lemma cdr-tv-dec-pass-length (rewrite)
  (equal (length (cdr (tv-dec-pass c a tree)))
         (tree-size tree))
  ;;Hint
  ((enable tv-dec-pass length tree-size)))


(prove-lemma tv-dec-pass-length (rewrite)
  (equal (length (tv-dec-pass c a tree))
         (add1 (tree-size tree)))
  ;;Hint
  ((enable length tv-dec-pass tree-size)))


(prove-lemma bvp-cdr-tv-dec-pass (rewrite)
  (bvp (cdr (tv-dec-pass c a tree)))
  ;;Hint
  ((enable bvp tv-dec-pass)))
```

```
(prove-lemma bvp-tv-dec-pass (rewrite)
  (bvp (tv-dec-pass c a tree))
  ;;Hint
  ((enable bvp tv-dec-pass)))


(prove-lemma bvp-length-tv-dec-pass (rewrite)
  (equal (bvp-length (tv-dec-pass c a tree) n)
         (leq n (add1 (tree-size tree))))
  ;;Hint
  ((enable bvp-length)))


(prove-lemma tv-dec-pass-crock-1 ()
  (implies
   (and (equal (length a) (tree-size tree))
        (bvp a))
   (equal (tv-adder c (v-not (nat-to-v 0 (length a))) a tree)
          (cons t (tv-dec-pass c a tree))))
  ;;Hint
  ((induct (tv-dec-pass c a tree))
   (disable v-not-firstn v-not-restn)
   (enable firstn-v-not restn-v-not t-carry)
   (expand (tv-dec-pass c a tree)
           (tv-adder c (v-not (nat-to-v 0 (length a))) a tree)
           (tv-dec-pass f a tree)
           (tv-adder f (v-not (nat-to-v 0 (length a))) a tree))))


(prove-lemma tv-dec-pass-crock-2 ()
  (implies
   (and (equal (length a) (tree-size tree))
        (bvp a))
   (equal (cdr (tv-dec-pass c a tree))
          (cdr (cdr (tv-adder c (v-not (nat-to-v 0 (length a))) a tree)))))
  ;;Hint
  ((use (tv-dec-pass-crock-1))))
```

```
(prove-lemma tv-dec-pass-works (rewrite)
  (implies
   (and (equal (length a) (tree-size tree))
        (bvp a))
   (equal (cdr (tv-dec-pass c a tree))
          (if c
              (v-buf a)
            (v-dec a))))
  ;;Hint
  ((use (tv-dec-pass-crock-2))
   (enable tv-adder-as-p-g-sum v-sum v-dec)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    TV-DEC-PASS-NG
;;;
;;;
;;;  The generate bit is of no consequence at the top level.  We want to
;;;  capture the specification of a dec/pass unit that only produces a
;;;  generate bit when necessary.  TV-DEC-PASS-NG only produces a generate bit
;;;  when MAKE-G is T.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn tv-dec-pass-ng (c a tree make-g)
  (if (nlistp tree)
      (if make-g
          (list (b-buf (car a)) (b-equv (car a) c))
        (list (b-equv (car a) c)))
    (let ((lhs (tv-dec-pass-ng c
                               (tfirstn a tree)
                               (car tree)
                               t)))
      (let ((rhs (tv-dec-pass-ng (b-or c (car lhs))
                                 (trestn a tree)
                                 (cdr tree)
                                 make-g)))
        (if make-g
            (cons (b-or (car lhs) (car rhs))
                  (append (cdr lhs) (cdr rhs)))
          (append (cdr lhs) rhs))))))


(disable tv-dec-pass-ng)
```

```
(prove-lemma tv-dec-pass-ng-length (rewrite)
  (equal (length (tv-dec-pass-ng c a tree make-g))
         (if make-g
             (add1 (tree-size tree))
           (tree-size tree)))
  ;;Hint
  ((enable tv-dec-pass-ng length tree-size)))


(prove-lemma tv-dec-pass-ng-length-1 (rewrite)
  (implies make-g
           (equal (length (cdr (tv-dec-pass-ng c a tree make-g)))
                  (tree-size tree)))
  ;;Hint
  ((enable tv-dec-pass-ng length tree-size)))


(prove-lemma bvp-tv-dec-pass-ng (rewrite)
  (bvp (tv-dec-pass-ng c a tree make-g))
  ;;Hint
  ((enable bvp tv-dec-pass-ng)))


(prove-lemma bvp-cdr-tv-dec-pass-ng (rewrite)
  (implies make-g
           (bvp (cdr (tv-dec-pass-ng c a tree make-g))))
  ;;Hint
  ((enable bvp tv-dec-pass-ng)))


(prove-lemma bvp-length-tv-dec-pass-ng (rewrite)
  (equal (bvp-length (tv-dec-pass-ng c a tree make-g) n)
         (if make-g
             (leq n (add1 (tree-size tree)))
           (lessp n (add1 (tree-size tree)))))
  ;;Hint
  ((enable bvp-length)))


(prove-lemma tv-dec-pass-ng-is-cdr-tv-dec-pass (rewrite)
  (equal (tv-dec-pass-ng c a tree make-g)
         (if make-g
             (tv-dec-pass c a tree)
           (cdr (tv-dec-pass c a tree))))
  ((enable tv-dec-pass tv-dec-pass-ng)
   (disable b-or tfirstn trestn)))


(prove-lemma boolp-car-tv-dec-pass-ng (rewrite)
  (boolp (car (tv-dec-pass-ng c a tree make-g)))
  ;;Hint
  ((enable tv-dec-pass-ng)))
```

```
(prove-lemma tv-dec-pass-ng-works-1 (rewrite)
  (implies
   (and (bvp a)
        (equal (length a)
               (tree-size tree))
        make-g)
   (equal (cdr (tv-dec-pass-ng c a tree make-g))
          (if c
              (v-buf a)
            (v-dec a)))))


(prove-lemma tv-dec-pass-ng-works-2 (rewrite)
  (implies
   (and (bvp a)
        (equal (length a)
               (tree-size tree))
        (not make-g))
   (equal (tv-dec-pass-ng c a tree make-g)
          (if c
              (v-buf a)
            (v-dec a)))))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    F$TV-DEC-PASS-NG
;;;
;;;    F$TV-DEC-PASS-NG is the 4-valued equivalent of TV-DEC-PASS-NG.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn f$tv-dec-pass-ng (c a tree make-g)
  (if (nlistp tree)
      (if make-g
          (list (car a) (f-equv (car a) c))
        (list (f-equv (car a) c)))
    (let ((lhs (f$tv-dec-pass-ng c
                                 (tfirstn a tree)
                                 (car tree)
                                 t)))
      (let ((rhs (f$tv-dec-pass-ng (f-or c (car lhs))
                                   (trestn a tree)
                                   (cdr tree)
                                   make-g)))
        (if make-g
            (cons (f-or (car lhs) (car rhs))
                  (append (cdr lhs) (cdr rhs)))
          (append (cdr lhs) rhs))))))


(disable f$tv-dec-pass-ng)
```

```
(prove-lemma f$tv-dec-pass-ng-length (rewrite)
  (equal (length (f$tv-dec-pass-ng c a tree make-g))
         (if make-g
             (add1 (tree-size tree))
           (tree-size tree)))
  ;;Hint
  ((enable f$tv-dec-pass-ng length tree-size)
   (disable-theory f-gates)))


(prove-lemma f$tv-dec-pass-ng-length-1 (rewrite)
  (implies make-g
           (equal (length (cdr (f$tv-dec-pass-ng c a tree make-g)))
                  (tree-size tree)))
  ;;Hint
  ((enable f$tv-dec-pass-ng length tree-size)
   (disable-theory f-gates)))


(prove-lemma properp-f$tv-dec-pass-ng (rewrite)
  (properp (f$tv-dec-pass-ng c a tree make-g))
  ;;Hint
  ((enable properp f$tv-dec-pass-ng)
   (disable-theory f-gates)))


(prove-lemma properp-cdr-f$tv-dec-pass-ng (rewrite)
  (implies make-g
           (properp (cdr (f$tv-dec-pass-ng c a tree make-g))))
  ;;Hint
  ((enable properp f$tv-dec-pass-ng)
   (disable-theory f-gates)))


(prove-lemma f$tv-dec-pass-ng=tv-dec-pass-ng$super-crock (rewrite)
  (implies
   (equal (f$tv-dec-pass-ng c a tree make-g)
          (tv-dec-pass-ng c a tree make-g))
   (boolp (car (f$tv-dec-pass-ng c a tree make-g)))))


(disable f$tv-dec-pass-ng=tv-dec-pass-ng$super-crock)
```

```
(prove-lemma f$tv-dec-pass-ng=tv-dec-pass-ng (rewrite)
  (implies
   (and (boolp c)
        (bvp a)
        (equal (length a) (tree-size tree)))
   (equal (f$tv-dec-pass-ng c a tree make-g)
          (tv-dec-pass-ng c a tree make-g)))
  ;;Hint
  ((induct (tv-dec-pass-ng c a tree make-g))
   (enable bvp boolp-b-gates b-buf-x=x
           f$tv-dec-pass-ng=tv-dec-pass-ng$super-crock)
   (expand (f$tv-dec-pass-ng c a tree make-g)
           (tv-dec-pass-ng c a tree make-g)
           (f$tv-dec-pass-ng c a tree f)
           (tv-dec-pass-ng c a tree f))
   (disable-theory f-gates b-gates)
   (disable tv-dec-pass-ng-is-cdr-tv-dec-pass
            tv-dec-pass-ng-works-1 tv-dec-pass-ng-works-2)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    TV-DEC-PASS-NG*
;;;
;;;    A module generator that implements TV-DEC-PASS-NG.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn dec-pass-cell* ()
  '(dec-pass-cell (c a) (g z)
                  ((g0 (g) id (a))
                   (g1 (z) b-equv (a c)))
                  nil))


(module-predicate dec-pass-cell*)


(module-netlist dec-pass-cell*)


(prove-lemma dec-pass-cell$value (rewrite)
  (implies
   (dec-pass-cell& netlist)
   (equal (dual-eval 0 'dec-pass-cell (list c a) state netlist)
          (list a (f-equv a c))))
  ;;Hint
  ((enable dec-pass-cell& id$value b-equv$value)
   (disable-theory f-gates)))
```

```
(defn tv-dec-pass-name (tree make-g)
  (if make-g
      #i(tv-dec-pass-g (tree-number tree))
      #i(tv-dec-pass-ng (tree-number tree))))

;;; Notice that while TV-DEC-PASS-NG has different specifications based on a
;;; flag, here we generate different circuits based on the flag, and
;;; therefore must give different names to modules that produce a generate
;;; and that do not produce a generate.


(defn tv-dec-pass-ng-body (tree make-g)
  (let ((a-names (indices 'a 0 (tree-size tree)))
        (z-names (indices 'z 0 (tree-size tree))))
    (let ((left-a-names (tfirstn a-names tree))
          (right-a-names (trestn a-names tree))
          (left-z-names (tfirstn z-names tree))
          (right-z-names (trestn z-names tree)))
      (if (nlistp tree)
          (list
           (list 'leaf (list 'g #i(z 0))
                 'dec-pass-cell (list 'c #i(a 0))))
        (if make-g
            (list
             (list 'left (cons 'gl left-z-names)
                   #i(tv-dec-pass-g (tree-number (car tree)))
                   (cons 'c left-a-names))
             (list 'carry '(cx) 'b-or '(c gl))
             (list 'right (cons 'gr right-z-names)
                   #i(tv-dec-pass-g (tree-number (cdr tree)))
                   (cons 'cx right-a-names))
             (list 'generate '(g) 'b-or '(gl gr)))
          (list
             (list 'left (cons 'gl left-z-names)
                   #i(tv-dec-pass-g (tree-number (car tree)))
                   (cons 'c left-a-names))
             (list 'carry '(cx) 'b-or '(c gl))
             (list 'right right-z-names
                   #i(tv-dec-pass-ng (tree-number (cdr tree)))
                   (cons 'cx right-a-names))))))))


(disable tv-dec-pass-ng-body)


(module-generator (tv-dec-pass-ng* tree make-g)
                  (tv-dec-pass-name tree make-g)
                  (cons 'c (indices 'a 0 (tree-size tree)))
                  (if make-g
                      (cons 'g (indices 'z 0 (tree-size tree)))
                    (indices 'z 0 (tree-size tree)))
                  (tv-dec-pass-ng-body tree make-g)
                  nil)
```

```
(defn tv-dec-pass-ng& (netlist tree make-g)
  (let ((current-tv-dec-pass (tv-dec-pass-name tree make-g)))
    (let ((xnetlist (delete-module current-tv-dec-pass netlist)))

      (if (nlistp tree)
          (and (equal (lookup-module current-tv-dec-pass netlist)
                      (tv-dec-pass-ng* tree make-g))
               (dec-pass-cell& xnetlist))

        (and (equal (lookup-module current-tv-dec-pass netlist)
                    (tv-dec-pass-ng* tree make-g))
             (tv-dec-pass-ng& xnetlist (car tree) t)
             (tv-dec-pass-ng& xnetlist (cdr tree) make-g)
             (b-or& xnetlist))))))


(disable tv-dec-pass-ng&)
```

```
(defn tv-dec-pass-ng-induction (tree c a make-g name state netlist)
  (let ((left-a (tfirstn a tree))
        (right-a (trestn a tree))
        (module-to-delete (tv-dec-pass-name tree make-g)))

    (if (nlistp tree)
        t

      (and
       (tv-dec-pass-ng-induction
        (car tree)
        c
        left-a
        t
        #i(tv-dec-pass-g (tree-number (car tree)))
        0
        (delete-module module-to-delete netlist))


       (tv-dec-pass-ng-induction
        (cdr tree)
        (f-or c (car (f$tv-dec-pass-ng c left-a (car tree) t)))
        right-a
        t
        #i(tv-dec-pass-g (tree-number (cdr tree)))
        0
        (delete-module module-to-delete netlist))

       (tv-dec-pass-ng-induction
        (cdr tree)
        (f-or c (car (f$tv-dec-pass-ng c left-a (car tree) t)))
        right-a
        f
        #i(tv-dec-pass-ng (tree-number (cdr tree)))
        0
        (delete-module module-to-delete netlist))))))
```

```
(prove-lemma tv-dec-pass-ng-lemma-crock (rewrite)
  (and
   (implies
    (tv-dec-pass-ng& (delete-module
                      #i(tv-dec-pass-g (tree-number tree))
                      netlist)
                     (car tree)
                     make-g)
    (equal (collect-value (indices 'z 0 n) bindings)
           (append (collect-value (firstn (tree-size (car tree))
                                          (indices 'z 0 n))
                                  bindings)
                   (collect-value (restn (tree-size (car tree))
                                         (indices 'z 0 n))
                                  bindings))))
   (implies
    (tv-dec-pass-ng& (delete-module
                      #i(tv-dec-pass-ng (tree-number tree))
                      netlist)
                     (car tree)
                     make-g)
    (equal (collect-value (indices 'z 0 n) bindings)
           (append (collect-value (firstn (tree-size (car tree))
                                          (indices 'z 0 n))
                                  bindings)
                   (collect-value (restn (tree-size (car tree))
                                         (indices 'z 0 n))
                                  bindings)))))
        ;;Hint
        ((use (collect-value-splitting-crock
               (l (indices 'z 0 n))
               (n (tree-size (car tree)))
               (alist bindings)))))


(disable tv-dec-pass-ng-lemma-crock)
```

```
(prove-lemma tv-dec-pass-ng$value (rewrite)
  (implies
   (and (tv-dec-pass-ng& netlist tree make-g)
        (equal (length a) (tree-size tree))
        (properp a)
        (boolp make-g)
        (equal name (tv-dec-pass-name tree make-g)))
   (equal (dual-eval 0
                     name
                     (cons c a) state netlist)
          (f$tv-dec-pass-ng c a tree make-g)))
  ;;Hint
  ((induct (tv-dec-pass-ng-induction tree c a make-g name state netlist))
   (disable open-indices indices)
   (enable tv-dec-pass-ng*$destructure
           tv-dec-pass-ng-lemma-crock
           dec-pass-cell$value b-or$value
           tv-dec-pass-ng-body boolp
           f$tv-dec-pass-ng tv-dec-pass-ng&)
   (disable-theory f-gates)
   (expand (tv-dec-pass-ng& netlist tree f)
           (tv-dec-pass-ng& netlist tree make-g))))


(disable tv-dec-pass-ng$value)


(defn tv-dec-pass-ng$netlist (tree make-g)
  (if (nlistp tree)
      (list (tv-dec-pass-ng* tree make-g)
            (dec-pass-cell*))
    (cons (tv-dec-pass-ng* tree make-g)
          (union (tv-dec-pass-ng$netlist (car tree) t)
                 (tv-dec-pass-ng$netlist (cdr tree) make-g)))))


(prove-lemma check-tv-dec-pass-ng$netlist ()
  (implies (boolp make-g)
           (tv-dec-pass-ng&
            (tv-dec-pass-ng$netlist '(0 . (0 . 0)) make-g)
            '(0 . (0 . 0))
            make-g))
  ((enable boolp)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    DEC-PASS*
;;;
;;;  If the control line C is high does a decrement, else passes A.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(module-generator (dec-pass* n)
                  #i(dec-pass n)
                  (cons 'c #i(a 0 n))
                  #i(z 0 n)
                  (list
                   (list 'm0 '(cn) 'b-not '(c))
                   (list 'm1
                         #i(z 0 n)
                         #i(tv-dec-pass-ng (tree-number
                                            (make-tree n)))
                         (cons 'cn #i(a 0 n))))
                  nil)


(defn dec-pass& (netlist n)
  (let ((xnetlist (delete-module #i(dec-pass n) netlist)))
    (and (equal (lookup-module #i(dec-pass n) netlist)
                (dec-pass* n))
         (b-not& xnetlist)
         (tv-dec-pass-ng& xnetlist (make-tree n) f))))


(disable dec-pass&)


(defn f$dec-pass (c a)
  (f$tv-dec-pass-ng (f-not c) a (make-tree (length a)) f))


(disable f$dec-pass)


(prove-lemma properp-length-f$dec-pass (rewrite)
  (and (properp (f$dec-pass c v))
       (implies
        (not (equal (length v) 0))
        (equal (length (f$dec-pass c v))
               (length v))))
  ;;Hint
  ((enable f$dec-pass)))


(prove-lemma f$dec-pass=dec-or-pass (rewrite)
  (implies
   (and (bvp v)
        (boolp dec)
        (not (equal (length v) 0)))
   (equal (f$dec-pass dec v)
          (if* dec (v-dec v) v)))
  ;;Hint
  ((enable f$dec-pass)))
```

```
(prove-lemma dec-pass$value (rewrite)
  (implies
   (and (dec-pass& netlist n)
        (not (zerop n))
        (equal (length a) n)
        (properp a))
   (equal (dual-eval 0 #i(dec-pass n) (cons c a) state netlist)
          (f$dec-pass c a)))
  ;;Hint
  ((enable dec-pass& dec-pass*$destructure b-not$value tv-dec-pass-ng$value
           f$dec-pass)
   (disable open-indices length)
   (disable-theory f-gates)))


(disable dec-pass$value)


(defn dec-pass$netlist (n)
  (cons (dec-pass* n)
        (union (b-not$netlist)
               (tv-dec-pass-ng$netlist (make-tree n) f))))


(prove-lemma check-dec-pass$netlist ()
  (dec-pass& (dec-pass$netlist 7) 7))
```

14.48 "reg.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    REG.EVENTS
;;;
;;;  Definitions and proofs of n-bit scan registers.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    REG* n
;;;
;;;  An n-bit scan register. Scans from low-order to high-order.  The
;;;  high order bit Q_(n-1) is the scan-out.
;;;
;;;  MODULE REG_n;
;;;  INPUTS CLK, TE, TI, D_0 ... D_(n-1);
;;;  OUTPUTS Q_0 ... Q_(n-1);
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn reg-body (m n ti te)
  (if (zerop n)
      nil
    (cons
     (list #i(G m)                      ;Occurrence name - G_m
           (list #i(Q m) #i(QB m))      ;Outputs (Q_m , QB_m)
           'FD1S                        ;Type FD1S
           (list #i(D m) 'CLK ti te))   ;Inputs
     (reg-body (add1 m) (sub1 n) #i(Q m) te))))


(disable reg-body)


(defn reg* (n)
  (list #i(REG n)                                     ;Name
        (list* 'CLK 'TE 'TI (indices 'D 0 n))         ;Inputs
        (indices 'Q 0 n)                              ;Outputs
        (list*
         (list 'TE-BUFFER '(TE-BUF)                   ;Body
               (if (lessp n 8) 'B-BUF 'B-BUF-PWR)
               '(TE))
         '(TI-DEL (TI-BUF) DEL4 (TI))
         (reg-body 0 n 'TI-BUF 'TE-BUF))
        (indices 'G 0 n)))             ;Statelist
```

```
(destructuring-lemma reg*)


(defn reg& (netlist n)
  (and (equal (lookup-module (index 'REG n) netlist)
              (reg* n))
       (let ((netlist (delete-module (index 'REG n) netlist)))
         (and (if (lessp n 8)
                  (b-buf& netlist)
                  (b-buf-pwr& netlist))
              (del4& netlist)
              (fd1s& netlist)))))


(disable reg&)


(defn reg$netlist (n)
  (cons (reg* n)
        (union (if (lessp n 8)
                   (b-buf$netlist)
                   (b-buf-pwr$netlist))
               (union (del4$netlist)
                      (fd1s$netlist)))))

;;;    REG value


(prove-lemma reg-body$unbound-in-body (rewrite)
  (and (unbound-in-body 'CLK (reg-body m n ti te))
       (unbound-in-body 'TI (reg-body m n ti te))
       (unbound-in-body 'TI-DEL (reg-body m n ti te))
       (unbound-in-body 'TE (reg-body m n ti te))
       (unbound-in-body 'TE-BUF (reg-body m n ti te))
       (unbound-in-body #i(D l) (reg-body m n ti te))
       (implies
        (lessp l m)
        (unbound-in-body (index 'Q l) (reg-body m n ti te))))
  ;;Hint
  ((enable unbound-in-body reg-body)))


(disable reg-body$unbound-in-body)


(prove-lemma reg-body$all-unbound-in-body (rewrite)
  (all-unbound-in-body (indices 'D x y) (reg-body m n ti te))
  ;;Hint
  ((enable all-unbound-in-body reg-body indices-as-append)))


(disable reg-body$all-unbound-in-body)
```

```
(defn reg-body-induction (m n ti te bindings state-bindings netlist)
  (if (zerop n)
      t
    (reg-body-induction
     (add1 m)
     (sub1 n)
     #i(Q m)
     te
     (dual-eval-body-bindings 1 (reg-body m n ti te)
                              bindings state-bindings netlist)
     state-bindings
     netlist)))


(prove-lemma reg-body$value ()
  (implies
   (fd1s& netlist)
   (equal (collect-value
           (indices 'Q m n)
           (dual-eval 1 (reg-body m n ti te) bindings state-bindings netlist))
          (v-threefix (collect-value (indices 'G m n) state-bindings))))
  ;;Hint
  ((induct (reg-body-induction m n ti te bindings
                               state-bindings netlist))
   (enable reg-body fd1s$value reg-body$unbound-in-body)
   (disable f-not threefix)))


(prove-lemma reg$value (rewrite)
  (implies
   (and (reg& netlist n)
        (equal (length state) n)
        (properp state))
   (equal (dual-eval 0 (index 'REG n) inputs state netlist)
          (v-threefix state)))
  ;;Hint
  ((use (reg-body$value
         (m 0)
         (ti 'TI-BUF)
         (te 'TE-BUF)
         (netlist (delete-module (index 'REG n) netlist))
         (bindings (cons
                    (cons 'TI-BUF (f-buf (caddr inputs)))
                    (cons
                     (cons 'TE-BUF (f-buf (cadr inputs)))
                     (pairlist (list* 'CLK 'TE 'TI (indices 'D 0 n))
                               inputs))))
         (state-bindings (pairstates (indices 'G 0 n) state))))
   (enable reg& reg*$destructure b-buf-pwr$value b-buf$value
           del4$value)
   (disable open-v-threefix open-indices)))
```

```
(disable reg$value)

;;;   REG state


(prove-lemma reg-body$state ()
  (implies
   (and (fd1s& netlist)
        (not (value te bindings)))
   (equal (collect-value
           (indices 'G m n)
           (dual-eval 3 (reg-body m n ti te) bindings
                      state-bindings netlist))
          (v-threefix (collect-value (indices 'D m n) bindings))))
  ;;Hint
  ((enable reg-body fd1s$state)
   (disable threefix)))


(prove-lemma reg$state (rewrite)
  (implies
   (and (reg& netlist n)
        (equal (length d) n)
        (properp d)
        (not te))
   (equal (dual-eval 2 (index 'REG n)
                     (list* clk te ti d)
                     state netlist)
          (v-threefix d)))
  ;;Hint
  ((use (reg-body$state
         (m 0)
         (ti 'TI-BUF)
         (te 'TE-BUF)
         (netlist (delete-module (index 'reg n) netlist))
         (bindings
          (dual-eval 1
                     (reg-body 0 n 'TI-BUF 'TE-BUF)
                     (pairlist (list* 'TI-BUF 'TE-BUF 'CLK 'TE 'TI
                                      (indices 'D 0 n))
                               (list* (f-buf ti)
                                      (f-buf te) clk te ti d))
                     (pairstates (indices 'G 0 n) state)
                     (delete-module (index 'REG n) netlist)))
         (state-bindings (pairstates (indices 'G 0 n) state))))
   (enable reg& reg-body$unbound-in-body
           reg-body$all-unbound-in-body
           reg*$destructure b-buf-pwr$value b-buf$value
           del4$value)
   (disable open-indices open-v-threefix v-threefix)))
```

```
(disable reg$state)


;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    WE-REG* n
;;;
;;;  An n-bit write-enabled scan register.  Scans from low-order to
;;;  high-order.  The high order bit Q_(n-1) is the scan-out.
;;;
;;;  MODULE WE-REG_n;
;;;  INPUTS CLK, WE, TE, TI, D_0 ... D_(n-1);
;;;  OUTPUTS Q_0 ... Q_(n-1);
;;;
;;;
;;;  The test input TI is buffered by a 4ns delay to avoid problems with clock
;;;  skew.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn we-reg-body (m n ti)
  (if (zerop n)
      nil
    (cons
     (list #i(G m)                              ;Occurrence name - G_m
           (list #i(Q m) #i(QB m))              ;Outputs (Q_m , QB_m)
           'FD1SLP                              ;Type FD1SLP
           (list #i(D m) 'CLK 'WE-BUF ti 'TE-BUF))      ;Inputs
     (we-reg-body (add1 m) (sub1 n) #i(Q m)))))


(disable we-reg-body)


(defn we-reg* (n)
  (list #i(WE-REG n)                                    ;Name
        (list* 'CLK 'WE 'TE 'TI (indices 'D 0 n))       ;Inputs
        (indices 'Q 0 n)                                ;Outputs
        (list*
         (list 'WE-BUFFER '(WE-BUF)                     ;Body
               (if (lessp n 8) 'B-BUF 'B-BUF-PWR)
               '(WE))
         (list 'TE-BUFFER '(TE-BUF)
               (if (lessp n 8) 'B-BUF 'B-BUF-PWR)
               '(TE))
         '(TI-DEL (TI-BUF) DEL4 (TI))
         (we-reg-body 0 n 'TI-BUF))
        (indices 'G 0 n)))              ;Statelist


(destructuring-lemma we-reg*)
```

```
(defn we-reg& (netlist n)
  (and (equal (lookup-module (index 'WE-REG n) netlist) (we-reg* n))
       (let ((netlist (delete-module (index 'WE-REG n) netlist)))
         (and (if (lessp n 8)
                  (b-buf& netlist)
                  (b-buf-pwr& netlist))
              (del4& netlist)
              (fd1slp& netlist)))))


(disable we-reg&)


(defn we-reg$netlist (n)
  (cons (we-reg* n)
        (union (if (lessp n 8)
                   (b-buf$netlist)
                   (b-buf-pwr$netlist))
               (union (del4$netlist)
                      (fd1slp$netlist)))))

;;;    WE-REG value


(prove-lemma we-reg-body$unbound-in-body (rewrite)
  (and (unbound-in-body 'CLK (we-reg-body m n ti))
       (unbound-in-body 'WE (we-reg-body m n ti))
       (unbound-in-body 'WE-BUF (we-reg-body m n ti))
       (unbound-in-body 'TI (we-reg-body m n ti))
       (unbound-in-body 'TI-BUF (we-reg-body m n ti))
       (unbound-in-body 'TE (we-reg-body m n ti))
       (unbound-in-body 'TE-BUF (we-reg-body m n ti))
       (unbound-in-body #i(D l) (we-reg-body m n ti))
       (implies
        (lessp l m)
        (unbound-in-body (index 'Q l) (we-reg-body m n ti))))
  ;;Hint
  ((enable unbound-in-body we-reg-body)))


(disable we-reg-body$unbound-in-body)


(prove-lemma we-reg-body$all-unbound-in-body (rewrite)
  (all-unbound-in-body (indices 'D x y) (we-reg-body m n ti))
  ;;Hint
  ((enable all-unbound-in-body we-reg-body indices-as-append)))


(disable we-reg-body$all-unbound-in-body)
```

```
(defn we-reg-body-induction (m n ti bindings state-bindings netlist)
  (if (zerop n)
      t
    (we-reg-body-induction
     (add1 m)
     (sub1 n)
     #i(Q m)
     (dual-eval-body-bindings 1 (we-reg-body m n ti)
                              bindings state-bindings netlist)
     state-bindings
     netlist)))


(prove-lemma we-reg-body$value (rewrite)
  (implies
   (fd1slp& netlist)
   (equal (collect-value
           (indices 'Q m n)
           (dual-eval 1 (we-reg-body m n ti)
                      bindings state-bindings netlist))
          (v-threefix (collect-value (indices 'G m n)
                                     state-bindings))))
  ;;Hint
  ((induct (we-reg-body-induction m n ti bindings
                                  state-bindings netlist))
   (enable we-reg-body fd1slp$value we-reg-body$unbound-in-body)
   (disable f-not threefix)))


(disable we-reg-body$value)


(prove-lemma we-reg$value (rewrite)
  (implies
   (and (we-reg& netlist n)
        (equal (length state) n)
        (properp state))
   (equal (dual-eval 0 (index 'WE-REG n) inputs state netlist)
          (v-threefix state)))
  ;;Hint
  ((enable we-reg& b-buf$value b-buf-pwr$value del4$value
           we-reg-body$value we-reg*$destructure)
   (disable open-indices)))


(disable we-reg$value)

;;;   WE-REG state
```

```
(prove-lemma we-reg-body$state (rewrite)
  (implies
   (and (fd1slp& netlist)
        (not (value 'TE-BUF bindings)))
   (equal (collect-value
           (indices 'G m n)
           (dual-eval 3 (we-reg-body m n ti)
                      bindings state-bindings netlist))
          (fv-if (value 'WE-BUF bindings)
                 (collect-value (indices 'D m n) bindings)
                 (collect-value (indices 'G m n) state-bindings))))
  ;;Hint
  ((enable we-reg-body fd1slp$state boolp fv-if)
   (disable threefix)))


(disable we-reg-body$state)


(prove-lemma we-reg$state (rewrite)
  (implies
   (and (we-reg& netlist n)
        (equal (length d) n)
        (properp d)
        (equal (length state) n)
        (properp state)
        (not te))
   (equal (dual-eval 2 (index 'WE-REG n)
                     (list* clk we te ti d)
                     state netlist)
          (fv-if we d state)))
  ;;Hint
  ((enable we-reg& we-reg-body$state boolp
           b-buf-pwr$value b-buf$value del4$value
           we-reg-body$value we-reg$value
           we-reg-body$unbound-in-body
           we-reg-body$all-unbound-in-body
           we-reg*$destructure
           fv-if-rewrite)
   (disable-theory f-gates)
   (disable open-indices open-v-threefix)))


(disable we-reg$state)
```

14.49 "alu-specs.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    ALU-SPECS.EVENTS -- The high level specification of the ALU.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;   CVZBV is used to construct all of the values returned by the ALU
;;;   specification V-ALU.


(defn cvzbv (carry overflow vector)
  (cons carry (cons overflow (cons (v-zerop vector) vector))))


(defn c  (cvzbv) (car cvzbv))

(defn v  (cvzbv) (cadr cvzbv))

(defn bv (cvzbv) (cdddr cvzbv))

(defn n  (cvzbv) (v-negp (bv cvzbv)))

(defn zb (cvzbv) (caddr cvzbv))


(disable c)

(disable v)

(disable n)

(disable zb)

(disable bv)


(prove-lemma boolp-n (rewrite)
  (implies
   (bvp (bv v))
   (boolp (n v)))
  ;;Hint
  ((enable n)))
```

```
(prove-lemma boolp-bvp-cvzbv (rewrite)
  (and
   (equal (boolp (c (cvzbv c v bv)))
          (boolp c))
   (equal (boolp (v (cvzbv c v bv)))
          (boolp v))
   (boolp (zb (cvzbv c v bv)))
   (equal (bvp (bv (cvzbv c v bv)))
          (bvp bv)))
  ;;Hint
  ((enable c v zb bv)))


(prove-lemma bvp-cvzbv ()
  (implies
   (and (boolp (c cvzbv))
        (boolp (v cvzbv))
        (boolp (zb cvzbv))
        (bvp (bv cvzbv)))
   (bvp cvzbv))
  ;;Hint
  ((enable c v zb bv)))



;;;  Specification abbreviations for V-ALU.


(defn cvzbv-v-ror (c a)
  (cvzbv (if (nlistp a) c (nth 0 a)) f (v-ror a c)))


(defn cvzbv-v-adder (c a b)
  (cvzbv (v-adder-carry-out c a b)
         (v-adder-overflowp c a b)
         (v-adder-output    c a b)))


(defn cvzbv-v-lsl (a)
  (cvzbv-v-adder f a a))


(defn cvzbv-v-subtracter (c a b)
  (cvzbv (v-subtracter-carry-out c a b)
         (v-subtracter-overflowp c a b)
         (v-subtracter-output    c a b)))


(defn cvzbv-inc (a)
  (cvzbv-v-adder t a (nat-to-v 0 (length a))))
```

```
(defn cvzbv-neg (a)
  (cvzbv-v-subtracter f a (nat-to-v 0 (length a))))


(defn cvzbv-dec (a)
  (cvzbv-v-subtracter t (nat-to-v 0 (length a)) a))


(defn cvzbv-v-not (a)
  (cvzbv f f (v-not a)))


(defn cvzbv-v-asr (a)
  (cvzbv (if (listp a) (nth 0 a) f) f (v-asr a)))


(defn cvzbv-v-lsr (a)
  (cvzbv (if (listp a) (nth 0 a) f) f (v-lsr a)))


;;;  V-ALU c a b op
;;;
;;;  The programmer's view of the ALU.


(defn v-alu (c a b op)
  (cond ((equal op #v0000) (cvzbv f f (v-buf a)))
        ((equal op #v0001) (cvzbv-inc a))
        ((equal op #v0010) (cvzbv-v-adder c a b))
        ((equal op #v0011) (cvzbv-v-adder f a b))
        ((equal op #v0100) (cvzbv-neg a))
        ((equal op #v0101) (cvzbv-dec a))
        ((equal op #v0110) (cvzbv-v-subtracter c a b))
        ((equal op #v0111) (cvzbv-v-subtracter f a b))
        ((equal op #v1000) (cvzbv-v-ror c a))
        ((equal op #v1001) (cvzbv-v-asr a))
        ((equal op #v1010) (cvzbv-v-lsr a))
        ((equal op #v1011) (cvzbv f f (v-xor a b)))
        ((equal op #v1100) (cvzbv f f (v-or  a b)))
        ((equal op #v1101) (cvzbv f f (v-and a b)))
        ((equal op #v1110) (cvzbv-v-not a))
        (t                 (cvzbv f f (v-buf a)))))


(disable v-alu)
```

```
(prove-lemma boolp-c-v-alu (rewrite)
  (implies
   (and (boolp c)
        (bvp a)
        (not (equal (length a) 0)))
   (boolp (c (v-alu c a b op))))
  ;;Hint
  ((enable v-alu)
   (disable cvzbv)))


(prove-lemma boolp-v-v-alu (rewrite)
  (implies
   (and (boolp c)
        (bvp a)
        (not (equal (length a) 0)))
   (boolp (v (v-alu c a b op))))
  ;;Hint
  ((enable v-alu)
   (disable cvzbv)))


(prove-lemma boolp-zb-v-alu (rewrite)
  (boolp (zb (v-alu c a b op)))
  ;;Hint
  ((enable v-alu)
   (disable cvzbv
            v-adder-carry-out v-adder-overflowp v-adder-output
            v-subtracter-carry-out v-subtracter-overflowp
            v-subtracter-output)))


(prove-lemma bvp-bv-v-alu (rewrite)
  (implies
   (bvp a)
   (bvp (bv (v-alu c a b op))))
  ;;Hint
  ((enable v-alu)
   (disable cvzbv
            v-adder-carry-out v-adder-overflowp v-adder-output
            v-subtracter-carry-out v-subtracter-overflowp
            b-not)))


(prove-lemma bvp-v-alu (rewrite)
  (implies
   (and (bvp a)
        (boolp c)
        (not (equal (length a) 0)))
   (bvp (v-alu c a b op)))
  ;;Hint
  ((use (bvp-cvzbv (cvzbv (v-alu c a b op))))))
```

```
(prove-lemma length-cvzbv-subtracter (rewrite)
  (equal (length (cvzbv-v-subtracter c a b))
         (add1 (add1 (add1 (length a))))))


(prove-lemma length-cvzbv-adder (rewrite)
  (equal (length (cvzbv-v-adder c a b))
         (add1 (add1 (add1 (length a))))))


(prove-lemma length-v-alu (rewrite)
  (equal (length (v-alu c a b op))
         (add1 (add1 (add1 (length a)))))
  ;;Hint
  ((enable v-alu)
   (disable cvzbv-v-subtracter)))


(prove-lemma bvp-length-bv (rewrite)
  (and (equal (length (bv x))
              (difference (length x) 3))
       (implies
        (and (bvp x)
             (leq 3 (length x)))
        (bvp (bv x))))
  ;;Hint
  ((enable bv)))



;;;  UNARY-OP-CODE-P op-code
;;;
;;;  Recognizes ALU op-codes which are unary operations on the A operand.
;;;  For unary ALU op-codes, the B operand is arbitrary.  We also define a
;;;  "1-argument" version of V-ALU which is equivalent to V-ALU when the
;;;  ALU op-code is unary.


(defn unary-op-code-p (op-code)
  (or (equal op-code #v0000)            ;Move
      (equal op-code #v0001)            ;Inc
      (equal op-code #v0100)            ;Neg
      (equal op-code #v0101)            ;Dec
      (equal op-code #v1000)            ;ROR
      (equal op-code #v1001)            ;ASR
      (equal op-code #v1010)            ;LSR
      (equal op-code #v1110)            ;Not
      (equal op-code #v1111)            ;Move-15
      ))
```

```
(disable unary-op-code-p)

;;;  V-ALU-1 op-code
;;;
;;;  The 1-arg ALU.


(defn v-alu-1 (c a op-code)
  (v-alu c a a op-code))


(disable v-alu-1)


(prove-lemma bvp-v-alu-1 (rewrite)
  (implies
   (and (bvp a)
        (boolp c)
        (not (equal (length a) 0)))
   (bvp (v-alu-1 c a op)))
  ;;Hint
  ((use (bvp-cvzbv (cvzbv (v-alu c a a op))))
   (enable v-alu-1)))


(prove-lemma length-v-alu-1 (rewrite)
  (equal (length (v-alu-1 c a op))
         (add1 (add1 (add1 (length a)))))
  ;;Hint
  ((enable v-alu-1)))


(prove-lemma unary-op-code-p->v-alu=v-alu-1 (rewrite)
  (implies
   (unary-op-code-p op-code)
   (equal (v-alu c a b op-code)
          (v-alu-1 c a op-code)))
  ;;Hint
  ((enable unary-op-code-p v-alu v-alu-1)))


(disable unary-op-code-p->v-alu=v-alu-1)



;;;    ALU-INC-OP
;;;    ALU-DEC-OP
;;;
;;;    These abbreviations are used for those cases where the processor ALU is
;;;    used for register pre-decrement and post-increment operations.


(defn alu-inc-op () #v0001)
```

```
(disable alu-inc-op)

(disable *1*alu-inc-op)


(prove-lemma bvp-length-alu-inc-op (rewrite)
  (and
   (equal (length (alu-inc-op)) 4)
   (bvp (alu-inc-op)))
  ;;Hint
  ((enable alu-inc-op)))


(defn alu-dec-op () #v0101)


(disable alu-dec-op)

(disable *1*alu-dec-op)


(prove-lemma bvp-length-alu-dec-op (rewrite)
  (and
   (equal (length (alu-dec-op)) 4)
   (bvp (alu-dec-op)))
  ;;Hint
  ((enable alu-dec-op)))


(prove-lemma bv-v-alu-alu-inc-alu-dec (rewrite)
  (and
   (equal (bv (v-alu c a b (alu-inc-op)))
          (v-inc a))
   (equal (bv (v-alu c a b (alu-dec-op)))
          (v-dec a)))
  ;;Hint
  ((enable bv v-alu v-inc v-dec alu-inc-op alu-dec-op)))
```

14.50 "pre-alu.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;   PRE-ALU.EVENTS
;;;
;;;   The ALU decoders, and CARRY-IN-HELP
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#|

  This table shows the settings of the mode, propagate, and generate lines for
each op-code that are computed by DECODE-MODE, DECODE-PROP, and DECODE-GEN
respectively.  Further, it shows carry-in, carry-out, and overflow requirements
that are implemented by CARRY-IN-HELP, CARRY-OUT-HELP, and OVERFLOW-HELP
respectively.   This table is valid assuming that the SWAP input to
DECODE-MODE and DECODE-GEN is true only when the op-code is either ALU-INC-OP
or ALU-DEC-OP.  (The SWAP input is normally F; it is set to T only in the cases
where the B operand is incremented or decremented.)  Also note that the control
logic must set the op-code to ALU-INC-OP whenever the ZERO flag is T.


 FM9001   Mode  Prop  Gen C-in  C-out  Ovfl  Comment
 opcode         aab   aab
 3210            n     nn

 0000     f     tff   ttf   x   f      f     Move
 0001     t     -f+   ttf   t   co     *     Increment
 0010     t     tft   ftt   c   co     *     AddC
 0011     t     tft   ftt   f   co     *     Add
 0100     t     ftf   ttf   t   ~co    *     Negation
 0101     t     ttf   f-+   f   ~co    *     Decrement
 0110     t     ftt   tft   ~c  ~co    *     SubB
 0111     t     ftt   tft   t   ~co    *     Sub
 1000     f     tff   ttf   c   c|a0   f     Move (ROR)
 1001     f     tff   ttf   x   a0     f     Move (ASR)
 1010     f     tff   ttf   x   a0     f     Move (LSR)
 1011     f     tff   fft   x   f      f     XOR
 1100     f     tft   ttf   x   f      f     Or
 1101     f     fff   ftt   x   f      f     And
 1110     f     ftf   ttf   x   f      f     Not
 1111     f     tff   ttf   x   f      f     Move

 ZERO     t     fff   fff   t   f      f     ZERO

+/- = swap/~swap

* -- Overflow equations.  a = A(n), b = B(n), out = out(n)

x110.a.~b.out + x110.~a.b.~out     Sub,Subb
```

```
1010.a.~out                       Dec
0010.a.out                        Neg
x100.a.b.~out + x100.~a.~b.out    Add,Addc
1000.~a.out                       Inc

|#

;;;  DECODE-MODE
;;;
;;;  Mode = T for arithmetic operations, and ZEROing.


(defn decode-mode (op0 op1 op2 op3)
  (b-nor (b-nor3 op0 op1 op2)
         op3))


(defn-to-module decode-mode)


(prove-lemma decode-mode$value-zero (rewrite)
  (implies
   (decode-mode& netlist)
   (equal (dual-eval 0 'decode-mode (alu-inc-op) state netlist)
          #v1))
  ;;Hint
  ((enable decode-mode& b-nor$value b-nor3$value alu-inc-op)
   (disable f-and)))


(disable decode-mode$value-zero)

;;;  DECODE-PROP == (LIST PB PAN PA)
```

```
(defn decode-prop (zero swap op0 op1 op2 op3)
  (let ((zero- (b-not zero))
        (swap- (b-not swap))
        (op0-  (b-not op0))
        (op1-  (b-not op1))
        (op2-  (b-not op2))
        (op3-  (b-not op3)))
    (let ((zerop (b-not zero-))
          (swap  (b-not swap-))
          (op0   (b-not op0-))
          (op1   (b-not op1-))
          (op2   (b-not op2-))
          (op3   (b-not op3-)))

      (list (b-nand3 (b-nand4 op0- op1- op2 op3)
                     (b-nand op1 op3-)
                     (b-nand3 op2- op3- swap))
            (b-nor op2- (b-nor op3- (b-nor op0 op1-)))
            (b-and (b-nand3 (b-nand op3 (b-equv op0 op1))
                            (b-nand op2- (b-nand swap op3-))
                            (b-nand4 op0 op1- op2 op3-))
                   zero-)))))


(defn-to-module decode-prop)


(prove-lemma decode-prop$value-zero (rewrite)
  (implies
   (decode-prop& netlist)
   (equal (dual-eval 0 'decode-prop (list* t f (alu-inc-op)) state
                     netlist)
          #v000))
  ;;Hint
  ((enable decode-prop& b-and$value b-not$value b-nand$value b-nand3$value
           b-nand4$value b-nor$value b-equv$value alu-inc-op
           f-and-rewrite f-not-rewrite)
   (disable-theory f-gates)))


(disable decode-prop$value-zero)

;;;  DECODE-GEN = (LIST GBN GAN GA)
```

```
(defn decode-gen (zero swap op0 op1 op2 op3)
  (let ((zero- (b-not zero))
        (swap- (b-not swap))
        (op0-  (b-not op0))
        (op1-  (b-not op1))
        (op2-  (b-not op2))
        (op3-  (b-not op3)))
    (let ((zero (b-not zero-))
          (swap (b-not swap-))
          (op0  (b-not op0-))
          (op1  (b-not op1-))
          (op2  (b-not op2-))
          (op3  (b-not op3-)))
      (list (b-nand3 (b-nand3 op0 op3 (b-xor op1 op2))
                     (b-nand3 op2 op3- (b-nand op1- swap-))
                     (b-nand3 op1 op2- op3-))
            (b-nor (b-nand (b-nand4 op0 op1 op2- op3)
                           (b-nand3 op2 op3- (b-nand op1- swap-)))
                   zero)
            (b-nor (b-nand3 (b-nand3 op0 op3 (b-xor op1 op2))
                            (b-nand3 op0 op1- op2)
                            (b-nand3 op1 op2- op3-))
                   zero)))))


(defn-to-module decode-gen)


(prove-lemma decode-gen$value-zero (rewrite)
  (implies
   (decode-gen& netlist)
   (equal (dual-eval 0 'decode-gen (list* t f (alu-inc-op)) state netlist)
          #v000))
  ;;Hint
  ((enable decode-gen& b-and$value b-not$value b-nor$value
           b-nand3$value b-nand$value b-xor$value
           f-and-rewrite f-not-rewrite f-nor f-or alu-inc-op)
   (disable-theory f-gates)))


(disable decode-gen$value-zero)

;;;  MPG == (LIST GBN GAN GA PB PAN PA M)
```

```
(defn f$mpg (zsop)
  (let ((zero (car zsop))
        (swap (cadr zsop))
        (op0 (caddr zsop))
        (op1 (cadddr zsop))
        (op2 (caddddr zsop))
        (op3 (cadddddr zsop)))
    (append (f$decode-gen zero swap op0 op1 op2 op3)
            (append (f$decode-prop zero swap op0 op1 op2 op3)
                    (list (f$decode-mode op0 op1 op2 op3))))))

(disable f$mpg)

(defn mpg (zsop)
  (let ((zero (car zsop))
        (swap (cadr zsop))
        (op0 (caddr zsop))
        (op1 (cadddr zsop))
        (op2 (caddddr zsop))
        (op3 (cadddddr zsop)))
    (append (decode-gen zero swap op0 op1 op2 op3)
            (append (decode-prop zero swap op0 op1 op2 op3)
                    (list (decode-mode op0 op1 op2 op3))))))

(disable mpg)

(prove-lemma f$mpg=mpg (rewrite)
  (implies
   (and (bvp zsop)
        (equal (length zsop) 6))
   (equal (f$mpg zsop)
          (mpg zsop)))
  ;;Hint
  ((enable f$mpg mpg bvp-length)))

(prove-lemma properp-length-f$mpg (rewrite)
  (and (properp (f$mpg zsop))
       (equal (length (f$mpg zsop)) 7))
  ;;Hint
  ((enable f$mpg f$decode-gen f$decode-prop f$decode-mode)
   (disable-theory f-gates)))

(prove-lemma length-mpg (rewrite)
  (equal (length (mpg zsop)) 7)
  ;;Hint
  ((enable mpg)))
```

```
(prove-lemma bvp-mpg (rewrite)
  (bvp (mpg zsop))
  ;;Hint
  ((enable mpg)))


(prove-lemma properp-mpg (rewrite)
  (properp (mpg zsop))
  ;;Hint
  ((enable mpg)))


(prove-lemma mpg-if-op-code (rewrite)
  (equal (mpg (cons a (cons b (if c d e))))
         (if c
             (mpg (cons a (cons b d)))
           (mpg (cons a (cons b e))))))


(prove-lemma mpg-zero (rewrite)
  (equal (mpg (cons t (cons f (alu-inc-op))))
         #v1000000)
  ;;Hint
  ((enable mpg alu-inc-op)))


(disable mpg-zero)


(defn mpg* ()
  '(mpg (zero swap op0 op1 op2 op3) (gbn gan ga pb pan pa mode)
        ((m (mode)       decode-mode (op0 op1 op2 op3))
         (p (pb pan pa)  decode-prop (zero swap op0 op1 op2 op3))
         (g (gbn gan ga) decode-gen  (zero swap op0 op1 op2 op3)))
        nil))


(module-predicate mpg*)


(module-netlist mpg*)
```

```
(prove-lemma mpg$value (rewrite)
  (implies
   (and (mpg& netlist)
        (properp zsop)
        (equal (length zsop) 6))
   (equal (dual-eval 0 'mpg zsop state netlist)
          (f$mpg zsop)))
  ;;Hint
  ((enable f$mpg mpg&
           decode-mode$value decode-prop$value decode-gen$value
           f$decode-mode f$decode-gen f$decode-prop)
   (disable-theory f-gates)))


(disable mpg$value)


(prove-lemma mpg$value-zero (rewrite)
  (implies
   (and (mpg& netlist)
        (equal zsop (list* t f (alu-inc-op))))
   (equal (dual-eval 0 'mpg zsop state netlist)
          #v1000000))
  ;;Hint
  ((enable mpg  mpg& decode-mode$value decode-prop$value
           decode-gen$value alu-inc-op)))


(disable mpg$value-zero)

;;;  CARRY-IN-HELP
```

```
(defn carry-in-help (czop)
  (let ((c   (car czop))
        (z   (cadr czop))
        (op0 (caddr czop))
        (op1 (cadddr czop))
        (op2 (caddddr czop))
        (op3 (cadddddr czop)))
    (let ((c-   (b-not c))
          (op0- (b-not op0))
          (op1- (b-not op1))
          (op2- (b-not op2))
          (op3- (b-not op3)))
      (let ((c   (b-not c-))
            (op0 (b-not op0-))
            (op1 (b-not op1-))
            (op2 (b-not op2-))
            (op3 (b-not op3-)))

        (b-or (b-nand3 (b-nand3 op1- op2- op3-)
                       (b-nand3 op0- op1- op2)
                       (b-nand3 op0 op1 op2))
              (b-nand3 (b-nand op3 c)
                       (b-nand3 op0- op2- c)
                       (b-nand3 op0- op2 c-)))))))
```

```
(disable carry-in-help)
```

```
(defn f$carry-in-help (czop)
  (let ((c   (car czop))
        (z   (cadr czop))
        (op0 (caddr czop))
        (op1 (cadddr czop))
        (op2 (caddddr czop))
        (op3 (cadddddr czop)))
    (let ((c-   (f-not c))
          (op0- (f-not op0))
          (op1- (f-not op1))
          (op2- (f-not op2))
          (op3- (f-not op3)))
      (let ((c   (f-not c-))
            (op0 (f-not op0-))
            (op1 (f-not op1-))
            (op2 (f-not op2-))
            (op3 (f-not op3-)))

        (f-or (f-nand3 (f-nand3 op1- op2- op3-)
                       (f-nand3 op0- op1- op2)
                       (f-nand3 op0 op1 op2))
              (f-nand3 (f-nand op3 c)
                       (f-nand3 op0- op2- c)
                       (f-nand3 op0- op2 c-)))))))
```

```
(disable f$carry-in-help)


(prove-lemma f$carry-in-help=carry-in-help (rewrite)
  (implies
   (and (bvp czop)
        (equal (length czop) 6))
   (equal (f$carry-in-help czop)
          (carry-in-help czop)))
  ;;Hint
  ((enable carry-in-help f$carry-in-help boolp-b-gates equal-length-add1)
   (disable-theory f-gates b-gates)))


(defn carry-in-help* ()
  '(carry-in-help (cin z op0in op1in op2in op3in) (cout)
                  ((g0  (c- c)     b-nbuf  (cin))
                   (g1  (op0- op0) b-nbuf  (op0in))
                   (g2  (op1- op1) b-nbuf  (op1in))
                   (g3  (op2- op2) b-nbuf  (op2in))
                   (g4  (op3- op3) b-nbuf  (op3in))
                   (g5  (s5)       b-nand3 (op1- op2- op3-))
                   (g6  (s6)       b-nand3 (op0- op1- op2))
                   (g7  (s7)       b-nand3 (op0 op1 op2))
                   (g8  (s8)       b-nand3 (s5 s6 s7))
                   (g9  (s9)       b-nand  (op3 c))
                   (g10 (s10)      b-nand3 (op0- op2- c))
                   (g11 (s11)      b-nand3 (op0- op2 c-))
                   (g12 (s12)      b-nand3 (s9 s10 s11))
                   (g13 (cout)     b-or    (s8 s12)))
                  nil))


(module-predicate carry-in-help*)


(module-netlist carry-in-help*)


(prove-lemma carry-in-help-zero (rewrite)
  (equal (carry-in-help (cons c (cons t (alu-inc-op))))
         t)
  ;;Hint
  ((enable carry-in-help alu-inc-op)))


(disable carry-in-help-zero)
```

```
(prove-lemma carry-in-help-if-op-code (rewrite)
  (equal (carry-in-help (cons a (cons b (if c d e))))
         (if c
             (carry-in-help (cons a (cons b d)))
           (carry-in-help (cons a (cons b e))))))


(prove-lemma carry-in-help$value (rewrite)
  (implies
   (carry-in-help& netlist)
   (equal (dual-eval 0 'carry-in-help czop state netlist)
          (list (f$carry-in-help czop))))
  ;;Hint
  ((enable f$carry-in-help carry-in-help&
           b-nbuf$value b-nand$value b-nand3$value b-nand4$value b-or$value
           equal-length-add1 f-not-f-not=f-buf)
   (disable-theory f-gates)))


(disable carry-in-help$value)


(prove-lemma carry-in-help$value-zero (rewrite)
  (implies
   (and (carry-in-help& netlist)
        (equal czop (list* c t (alu-inc-op))))
   (equal (dual-eval 0 'carry-in-help czop state netlist)
          #v1))
  ;;Hint
  ((enable f$carry-in-help carry-in-help&
           b-nbuf$value b-nand$value b-nand3$value b-nand4$value b-or$value
           equal-length-add1 f-not-f-not=f-buf alu-inc-op
           boolp-b-gates)
   (disable-theory f-gates)))


(disable carry-in-help$value-zero)
```

14.51 "tv-alu-help.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    TV-ALU-HELP.EVENTS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  P-CELL


(defn p-cell (a an b pa pan pb)
  (b-nand3 (b-nand a pa)
           (b-nand an pan)
           (b-nand b pb)))


(defn-to-module p-cell)


(prove-lemma p-cell$value-zero (rewrite)
  (implies
   (p-cell& netlist)
   (equal (dual-eval 0 'p-cell (list a an b f f f) state netlist)
          (list f)))
  ;;Hint
  ((enable p-cell& b-nand3$value b-nand$value)))

;;;  G-CELL


(defn g-cell (a an bn ga gan gbn)
  (b-and3 (b-nand a ga)
          (b-nand an gan)
          (b-nand bn gbn)))


(defn-to-module g-cell)


(prove-lemma g-cell$value-zero (rewrite)
  (implies
   (g-cell& netlist)
   (equal (dual-eval 0 'g-cell (list a an bn f f f) state netlist)
          (list t)))
  ;;Hint
  ((enable g-cell& b-and3$value b-nand$value)))

;;;   ALU-CELL
```

```
(defn alu-cell (c a b mpg)
  (let ((gbn (car mpg))
        (gan (cadr mpg))
        (ga  (caddr mpg))
        (pb  (cadddr mpg))
        (pan (caddddr mpg))
        (pa  (cadddddr mpg))
        (m   (caddddddr mpg)))
    (let ((an (b-not a))
          (bn (b-not b)))
      (let ((p (p-cell a an b pa pan pb))
            (g (g-cell a an bn ga gan gbn))
            (mc (b-nand c m)))
        (let ((z (b-equv3 mc p g)))
          (list p g z))))))


(defn f$alu-cell (c a b mpg)
  (let ((gbn (car mpg))
        (gan (cadr mpg))
        (ga  (caddr mpg))
        (pb  (cadddr mpg))
        (pan (caddddr mpg))
        (pa  (cadddddr mpg))
        (m   (caddddddr mpg)))
    (let ((an (f-not a))
          (bn (f-not b)))
      (let ((p (f$p-cell a an b pa pan pb))
            (g (f$g-cell a an bn ga gan gbn))
            (mc (f-nand c m)))
        (let ((z (f-equv3 mc p g)))
          (list p g z))))))


(disable f$alu-cell)


(prove-lemma f$alu-cell=alu-cell (rewrite)
  (implies
   (and (boolp c)
        (boolp a)
        (boolp b)
        (bvp mpg)
        (equal (length mpg) 7))
   (equal (f$alu-cell c a b mpg)
          (alu-cell c a b mpg)))
  ;;Hint
  ((enable f$alu-cell bvp-length boolp-b-gates)
   (disable-theory f-gates b-gates)))
```

```
(defn alu-cell* ()
  '(alu-cell (c a b gbn gan ga pb pan pa m)
             (p g z)
             ((n0 (an) b-not (a))
              (n1 (bn) b-not (b))
              (p0 (p)  p-cell (a an b pa pan pb))
              (g0 (g)  g-cell (a an bn ga gan gbn))
              (m0 (mc) b-nand (c m))
              (z0 (z)  b-equv3 (mc p g)))
             nil))


(module-predicate alu-cell*)


(module-netlist alu-cell*)


(prove-lemma alu-cell$value (rewrite)
  (implies
   (alu-cell& netlist)
   (equal (dual-eval 0 'alu-cell (list* c a b mpg) state netlist)
          (f$alu-cell c a b mpg)))
  ;;Hint
  ((enable f$alu-cell alu-cell& b-not$value p-cell$value g-cell$value
           b-nand$value b-equv3$value bvp-length)
   (disable-theory f-gates)))


(disable alu-cell$value)


(prove-lemma alu-cell$value-zero (rewrite)
  (implies
   (and (alu-cell& netlist)
        (equal mpg #v1000000))
   (equal (dual-eval 0 'alu-cell (list* t a b mpg) state netlist)
          (list f t f)))
  ;;Hint
  ((enable alu-cell& bvp-length
           p-cell$value-zero g-cell$value-zero b-nand$value b-equv3$value)))


(disable alu-cell$value-zero)
```

```
(prove-lemma f$alu-cell-v-threefix-mpg (rewrite)
  (implies
   (equal (length mpg) 7)
   (equal (f$alu-cell c a b (v-threefix mpg))
          (f$alu-cell c a b mpg)))
  ;;Hint
  ((enable f$alu-cell f$p-cell f$g-cell v-threefix equal-length-add1
           f-gate-threefix-congruence-lemmas)
   (disable threefix)
   (disable-theory f-gates)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;  TV-ALU-HELP
;;;
;;;  The Boolean specification of the main component of the ALU.  The TREE
;;;  specifies the structure of the carry-lookahead tree of the ALU.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn tv-alu-help (c a b mpg tree)
  (if (nlistp tree)
      (alu-cell c (car a) (car b) mpg)
    (let ((a-car (tfirstn a tree))
          (b-car (tfirstn b tree))
          (a-cdr (trestn  a tree))
          (b-cdr (trestn  b tree)))
      (let ((lhs (tv-alu-help c a-car b-car mpg (car tree))))
        (let ((p-car (car lhs))
              (g-car (cadr lhs))
              (sum-car (cddr lhs)))
          (let ((c-car (t-carry c p-car g-car)))
            (let ((rhs (tv-alu-help c-car a-cdr b-cdr mpg (cdr tree))))
              (let ((p-cdr (car rhs))
                    (g-cdr (cadr rhs))
                    (sum-cdr (cddr rhs)))
                (cons (b-and p-car p-cdr)
                      (cons (t-carry g-car p-cdr g-cdr)
                            (append sum-car sum-cdr)))))))))))


(disable tv-alu-help)


(prove-lemma cddr-tv-alu-help-length (rewrite)
  (equal (length (cddr (tv-alu-help c a b mpg tree)))
         (tree-size tree))
  ;;Hint
  ((enable tv-alu-help length tree-size)))
```

```
(prove-lemma tv-alu-help-length (rewrite)
  (equal (length (tv-alu-help c a b mpg tree))
         (add1 (add1 (tree-size tree))))
  ;;Hint
  ((enable length tv-alu-help tree-size)))


(prove-lemma bvp-cddr-tv-alu-help (rewrite)
  (bvp (cddr (tv-alu-help c a b mpg tree)))
  ;;Hint
  ((enable bvp tv-alu-help)))


(prove-lemma bvp-tv-alu-help (rewrite)
  (bvp (tv-alu-help c a b mpg tree))
  ;;Hint
  ((enable bvp tv-alu-help)))


(prove-lemma bvp-length-tv-alu-help (rewrite)
  (equal (bvp-length (tv-alu-help c a b mpg tree) n)
         (leq n (add1 (add1 (tree-size tree)))))
  ;;Hint
  ((enable bvp-length)))

;;;  Proofs that TV-ALU-HELP "does the right thing."

;;;  ZERO


(prove-lemma tv-alu-help-zero (rewrite)
  (implies
   (and (equal mpg #v1000000)
        (equal (length a) (tree-size tree))
        (equal c t))
   (equal (tv-alu-help c a b mpg tree)
          (list* f t (make-list (length a) f))))
  ;;Hint
  ((induct (tv-alu-help c a b mpg tree))
   (enable tv-alu-help make-list make-list-append-tree-crock)
   (expand (tv-alu-help t a b #v1000000 tree))))


(disable tv-alu-help-zero)

;;;  V-AND
```

```
(prove-lemma tv-alu-help-v-and-works (rewrite)
  (implies (and (bv2p a b)
                (equal (length a) (tree-size tree))
                (equal mpg #v0000011))
           (equal (cddr (tv-alu-help c a b mpg tree))
                  (v-and a b)))
  ((induct (tv-alu-help c a b mpg tree))
   (expand (tv-alu-help c a b #v0000011 tree))
   (enable v-and)))


(disable tv-alu-help-v-and-works)

;;;  V-OR


(prove-lemma tv-alu-help-v-or-works (rewrite)
  (implies (and (equal mpg #v0101110)
                (bv2p a b)
                (equal (length a) (tree-size tree)))
           (equal (cddr (tv-alu-help c a b mpg tree))
                  (v-or a b)))
  ((induct (tv-alu-help c a b mpg tree))
   (expand (tv-alu-help c a b #v0101110 tree))
   (enable v-or)))


(disable tv-alu-help-v-or-works)

;;;  V-XOR


(prove-lemma tv-alu-help-v-xor-works (rewrite)
  (implies (and (equal mpg #v0100001)
                (bv2p a b)
                (equal (length a) (tree-size tree)))
           (equal (cddr (tv-alu-help c a b mpg tree))
                  (v-xor a b)))
  ((induct (tv-alu-help c a b mpg tree))
   (expand (tv-alu-help c a b #v0100001 tree))
   (enable v-xor)))


(disable tv-alu-help-v-xor-works)

;;;  V-NOT
```

```
(prove-lemma tv-alu-help-v-not-works (rewrite)
  (implies (and (equal mpg #v0010110)
                (bvp a)
                (equal (length a) (tree-size tree)))
           (equal (cddr (tv-alu-help c a b mpg tree))
                  (v-not a)))
  ((induct (tv-alu-help c a b mpg tree))
   (expand (tv-alu-help c a b #v0010110 tree))
   (enable v-not)))


(disable tv-alu-help-v-not-works)

;;;  V-BUF


(prove-lemma tv-alu-help-v-buf-works (rewrite)
  (implies (and (equal mpg #v0100110)
                (bvp a)
                (equal (length a) (tree-size tree)))
           (equal (cddr (tv-alu-help c a b mpg tree))
                  (v-buf a)))
  ((induct (tv-alu-help c a b mpg tree))
   (expand (tv-alu-help c a b #v0100110 tree))
   (enable v-buf)))


(disable tv-alu-help-v-buf-works)

;;;  TV-ADDER


(prove-lemma tv-alu-help-tv-adder-works (rewrite)
  (implies (and (equal mpg #v1101011)
                (bvp a)
                (equal (length a) (tree-size tree)))
           (equal (tv-alu-help c a b mpg tree)
                  (tv-adder c a b tree)))
  ((induct (tv-alu-help c a b mpg tree))
   (expand (tv-alu-help c a b #v1101011 tree)
           (tv-adder c a b tree))))


(disable tv-alu-help-tv-adder-works)

;;;  Subtracter
```

```
(prove-lemma tv-alu-help-tv-subtracter-works (rewrite)
  (implies (and (equal mpg #v1011101)
                (bvp a)
                (equal (length a) (tree-size tree)))
           (equal (tv-alu-help c a b mpg tree)
                  (tv-adder c (v-not a) b tree)))
  ((induct (tv-alu-help c a b mpg tree))
   (expand (tv-alu-help c a b #v1011101 tree)
           (tv-adder c (v-not a) b tree))
   (enable v-not)))


(disable tv-alu-help-tv-subtracter-works)

;;;  INC a


(prove-lemma tv-alu-help-tv-inc-a-works (rewrite)
  (implies (and (equal mpg #v1100110)
                (bvp a)
                (equal (length a) (tree-size tree)))
           (equal (tv-alu-help c a b mpg tree)
                  (tv-adder c a (nat-to-v 0 (length a)) tree)))
  ((induct (tv-alu-help c a b mpg tree))
   (expand (tv-alu-help c a b #v1100110 tree)
           (tv-adder c a (nat-to-v 0 (length a)) tree))))


(disable tv-alu-help-tv-inc-a-works)

;;;  INC b


(prove-lemma tv-alu-help-tv-inc-b-works (rewrite)
  (implies (and (equal mpg #v1001110)
                (bvp b)
                (equal (length b) (tree-size tree)))
           (equal (tv-alu-help c a b mpg tree)
                  (tv-adder c b (nat-to-v 0 (length b)) tree)))
  ((induct (tv-alu-help c a b mpg tree))
   (expand (tv-alu-help c a b #v1001110 tree)
           (tv-adder c b (nat-to-v 0 (length b)) tree))))


(disable tv-alu-help-tv-inc-b-works)

;;;  DEC a
```

```
(prove-lemma tv-alu-help-tv-dec-a-works (rewrite)
  (implies (and (equal mpg #v1110010)
                (bvp a)
                (equal (length a) (tree-size tree)))
           (equal (tv-alu-help c a b mpg tree)
                  (tv-adder c (v-not (nat-to-v 0 (length a))) a tree)))
  ((induct (tv-alu-help c a b mpg tree))
   (disable v-not-firstn v-not-restn)
   (enable firstn-v-not restn-v-not)
   (expand (tv-alu-help c a b #v1110010 tree)
           (tv-adder c (v-not (nat-to-v 0 (length a))) a tree))))


(disable tv-alu-help-tv-dec-a-works)

;;;  DEC b


(prove-lemma tv-alu-help-tv-dec-b-works (rewrite)
  (implies (and (equal mpg #v1110001)
                (bvp b)
                (equal (length b) (tree-size tree)))
           (equal (tv-alu-help c a b mpg tree)
                  (tv-adder c (v-not (nat-to-v 0 (length b))) b tree)))
  ((induct (tv-alu-help c a b mpg tree))
   (disable v-not-firstn v-not-restn)
   (enable firstn-v-not restn-v-not)
   (expand (tv-alu-help c a b #v1110001 tree)
           (tv-adder c (v-not (nat-to-v 0 (length b))) b tree))))


(disable tv-alu-help-tv-dec-b-works)

;;;  NEG


(prove-lemma tv-alu-help-tv-neg-works (rewrite)
  (implies (and (equal mpg #v1010110)
                (bvp a)
                (equal (length a) (tree-size tree)))
           (equal (tv-alu-help c a b mpg tree)
                  (tv-adder c (v-not a) (nat-to-v 0 (length a)) tree)))
  ((induct (tv-alu-help c a b mpg tree))
   (expand (tv-alu-help c a b #v1010110 tree)
           (tv-adder c (v-not a) (nat-to-v 0 (length a)) tree))
   (enable v-not)))
```

```
(disable tv-alu-help-tv-neg-works)

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;   F$TV-ALU-HELP
;;;
;;;   The 4-valued ALU specification equivalent to TV-ALU-HELP.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn f$tv-alu-help (c a b mpg tree)
  (if (nlistp tree)
      (f$alu-cell c (car a) (car b) mpg)
    (let ((a-car (tfirstn a tree))
          (b-car (tfirstn b tree))
          (a-cdr (trestn  a tree))
          (b-cdr (trestn  b tree)))
      (let ((lhs (f$tv-alu-help c a-car b-car mpg (car tree))))
        (let ((p-car (car lhs))
              (g-car (cadr lhs))
              (sum-car (cddr lhs)))
          (let ((c-car (f$t-carry c p-car g-car)))
            (let ((rhs (f$tv-alu-help c-car a-cdr b-cdr mpg (cdr tree))))
              (let ((p-cdr (car rhs))
                    (g-cdr (cadr rhs))
                    (sum-cdr (cddr rhs)))
                (cons (f-and p-car p-cdr)
                      (cons (f$t-carry g-car p-cdr g-cdr)
                            (append sum-car sum-cdr))))))))))


(disable f$tv-alu-help)


(prove-lemma cddr-f$tv-alu-help-length (rewrite)
  (equal (length (cddr (f$tv-alu-help c a b mpg tree)))
         (tree-size tree))
  ;;Hint
  ((enable f$tv-alu-help f$alu-cell length tree-size)))


(prove-lemma f$tv-alu-help-length (rewrite)
  (equal (length (f$tv-alu-help c a b mpg tree))
         (add1 (add1 (tree-size tree))))
  ;;Hint
  ((enable length f$tv-alu-help f$alu-cell tree-size)))
```

```
(prove-lemma properp-cddr-f$tv-alu-help (rewrite)
  (properp (cddr (f$tv-alu-help c a b mpg tree)))
  ;;Hint
  ((enable properp f$alu-cell)
   (induct (f$tv-alu-help c a b mpg tree))
   (expand (f$tv-alu-help c a b mpg tree))
   (disable-theory f-gates b-gates)))


(prove-lemma properp-f$tv-alu-help (rewrite)
   (properp (f$tv-alu-help c a b mpg tree))
  ;;Hint
  ((enable properp f$alu-cell)
   (expand (f$tv-alu-help c a b mpg tree))
   (disable-theory f-gates b-gates)))


(prove-lemma boolp-car-f$tv-alu-help (rewrite)
  (implies
   (and (boolp c)
        (bvp a) (equal (length a) (tree-size tree))
        (bvp b) (equal (length b) (tree-size tree))
        (bvp mpg) (equal (length mpg) 7))
   (and (boolp (car (f$tv-alu-help c a b mpg tree)))
        (boolp (cadr (f$tv-alu-help c a b mpg tree)))))
  ;;Hint
  ((induct (f$tv-alu-help c a b mpg tree))
   (expand (f$tv-alu-help c a b mpg tree))
   (disable-theory f-gates b-gates)
   (enable boolp-b-gates)))


(prove-lemma bvp-cddr-f$tv-alu-help (rewrite)
  (implies
   (and (boolp c)
        (bvp a) (equal (length a) (tree-size tree))
        (bvp b) (equal (length b) (tree-size tree))
        (bvp mpg) (equal (length mpg) 7))
   (bvp (cddr (f$tv-alu-help c a b mpg tree))))
  ;;Hint
  ((enable bvp boolp-b-gates)
   (induct (f$tv-alu-help c a b mpg tree))
   (expand (f$tv-alu-help c a b mpg tree))
   (disable-theory f-gates b-gates)))
```

```
(prove-lemma bvp-f$tv-alu-help (rewrite)
  (implies
   (and (boolp c)
        (bvp a) (equal (length a) (tree-size tree))
        (bvp b) (equal (length b) (tree-size tree))
        (bvp mpg) (equal (length mpg) 7))
   (bvp (f$tv-alu-help c a b mpg tree)))
  ;;Hint
  ((enable bvp boolp-b-gates)
   (expand (f$tv-alu-help c a b mpg tree))
   (disable-theory f-gates b-gates)))


(prove-lemma f$tv-alu-help=tv-alu-help (rewrite)
  (implies
   (and (boolp c)
        (bvp a) (equal (length a) (tree-size tree))
        (bvp b) (equal (length b) (tree-size tree))
        (bvp mpg) (equal (length mpg) 7))
   (equal (f$tv-alu-help c a b mpg tree)
          (tv-alu-help c a b mpg tree)))
  ;;Hint
  ((induct (tv-alu-help c a b mpg tree))
   (enable boolp-b-gates bvp-length)
   (expand (f$tv-alu-help c a b mpg tree)
           (tv-alu-help c a b mpg tree))
   (disable-theory f-gates b-gates)
   (disable alu-cell mpg t-carry)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;   TV-ALU-HELP-BODY
;;;
;;;   The hardware version of TV-ALU-HELP.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

;;;  TV-ALU-HELP-BODY
```

```
(defn tv-alu-help-body (tree)
  (let ((a-names (indices 'a 0 (tree-size tree)))
        (b-names (indices 'b 0 (tree-size tree)))
        (out-names (indices 'out 0 (tree-size tree)))
        (mpgnames (indices 'mpg 0 7))
        (mpgnames- (indices 'mpg- 0 7)))
    (let ((left-a-names (tfirstn a-names tree))
          (right-a-names (trestn a-names tree))
          (left-b-names (tfirstn b-names tree))
          (right-b-names (trestn b-names tree))
          (left-out-names (tfirstn out-names tree))
          (right-out-names (trestn out-names tree)))
      (let ((buffer? (equal (remainder (sub1 (tree-height tree)) 3) 0)))
        (let ((mpgnames? (if buffer? mpgnames- mpgnames)))
          (if (nlistp tree)
              (list
               (list 'leaf
                     (list 'p 'g (index 'out 0))
                     'alu-cell
                     (cons 'c (cons (index 'a 0)
                                    (cons (index 'b 0) mpgnames)))))
            ;;  We buffer MPG whenever appropriate.
            (append
             (if buffer?
                 (list
                  (list 'buffermpg mpgnames- (index 'v-buf 7) mpgnames))
               nil)
             (list
              ;;  The LHS ALU result.
              (list 'lhs
                    (cons 'pl (cons 'gl left-out-names))
                    (index 'tv-alu-help (tree-number (car tree)))
                    (cons 'c
                          (append left-a-names
                                  (append left-b-names
                                          mpgnames?))))
              ;;  The LHS carry.
              '(lhs-carry (cl) t-carry (c pl gl))
              ;;  The RHS ALU result.
              (list 'rhs
                    (cons 'pr (cons 'gr right-out-names))
                    (index 'tv-alu-help (tree-number (cdr tree)))
                    (cons 'cl
                          (append right-a-names
                                  (append right-b-names
                                          mpgnames?))))
              ;;  The propagate output.
              '(p (p) b-and (pl pr))
              ;;  The generate output.
              '(g (g) t-carry (gl pr gr))))))))))
```

```
(defn tv-alu-help* (tree)
  (let ((a-names (indices 'a 0 (tree-size tree)))
        (b-names (indices 'b 0 (tree-size tree)))
        (out-names (indices 'out 0 (tree-size tree)))
        (mpgnames (indices 'mpg 0 7)))
      (list
       ;;  Name
       (index 'tv-alu-help (tree-number tree))
       ;;  Inputs
       (cons 'c (append a-names (append b-names mpgnames)))
       ;;  Outputs
       (cons 'p (cons 'g out-names))
       ;;  Occurrences
       (tv-alu-help-body tree)
       ;;  States
       nil)))
```

```
(destructuring-lemma tv-alu-help*)
```

```
(defn tv-alu-help& (netlist tree)
  (if (nlistp tree)
      (and (equal (lookup-module (index 'tv-alu-help (tree-number tree))
                                 netlist)
                  (tv-alu-help* tree))
           (alu-cell& (delete-module (index 'tv-alu-help (tree-number tree))
                                     netlist)))
    (and (equal (lookup-module (index 'tv-alu-help (tree-number tree)) netlist)
                (tv-alu-help* tree))
         (t-carry& (delete-module (index 'tv-alu-help (tree-number tree))
                                  netlist))
         (b-and& (delete-module (index 'tv-alu-help (tree-number tree))
                                netlist))
         (v-buf& (delete-module (index 'tv-alu-help (tree-number tree))
                                netlist)
                 7)
         (tv-alu-help& (delete-module (index 'tv-alu-help (tree-number tree))
                                      netlist)
                       (car tree))
         (tv-alu-help& (delete-module (index 'tv-alu-help (tree-number tree))
                                      netlist)
                       (cdr tree)))))
```

```
(disable tv-alu-help&)
```

```
(defn tv-alu-help$netlist (tree)
  (if (nlistp tree)
      (cons (tv-alu-help* tree) (alu-cell$netlist))
    (cons (tv-alu-help* tree)
          (union (tv-alu-help$netlist (car tree))
                 (union (tv-alu-help$netlist (cdr tree))
                        (union (t-carry$netlist)
                               (union (b-and$netlist)
                                      (v-buf$netlist 7))))))))


(defn tv-alu-help-induction (tree c a b mpg state netlist)
  (let ((left-a (tfirstn a tree))
        (left-b (tfirstn b tree))
        (right-a (trestn a tree))
        (right-b (trestn b tree))
        (buffer? (equal (remainder (sub1 (tree-height tree)) 3) 0)))
    (let ((mpg (if buffer? (v-threefix mpg) mpg)))
      (if (nlistp tree)
          t
        (and
         (tv-alu-help-induction
          (car tree)
          c
          left-a
          left-b
          mpg
          0
          (delete-module (index 'tv-alu-help (tree-number tree)) netlist))
         (tv-alu-help-induction
          (cdr tree)
          (f$t-carry c
                     (car (f$tv-alu-help c left-a left-b mpg (car tree)))
                     (cadr (f$tv-alu-help c left-a left-b mpg (car tree))))
          right-a
          right-b
          mpg
          0
          (delete-module (index 'tv-alu-help (tree-number tree)) netlist)))))))
```

```
(prove-lemma tv-alu-help-lemma-crock (rewrite)
  (implies
   (tv-alu-help& (delete-module (index 'tv-alu-help (tree-number tree))
                                netlist)
                 (car tree))
   (equal (collect-value (indices 'out 0 n) bindings)
          (append (collect-value (firstn (tree-size (car tree))
                                         (indices 'out 0 n))
                                 bindings)
                  (collect-value (restn (tree-size (car tree))
                                        (indices 'out 0 n))
                                 bindings))))
  ;;Hint
  ((use (collect-value-splitting-crock
         (l (indices 'out 0 n))
         (n (tree-size (car tree)))
         (alist bindings)))))


(disable tv-alu-help-lemma-crock)


(prove-lemma tv-alu-help$value-base-case (rewrite)
  (implies
   (and (nlistp tree)
        (tv-alu-help& netlist tree)
        (equal (length a) 1) (properp a)
        (equal (length b) 1) (properp b)
        (equal (length mpg) 7) (properp mpg))
   (equal (dual-eval 0 (index 'tv-alu-help (tree-number tree))
                     (cons c (append a (append b mpg)))
                     state netlist)
          (f$tv-alu-help c a b mpg tree)))
  ;;Hint
  ((disable f-and *1*indices open-indices)
   (enable  alu-cell$value f$alu-cell tv-alu-help*$destructure)
   (expand (tv-alu-help& netlist tree)
           (f$tv-alu-help c a b mpg tree)
           (append b mpg)
           (append a (cons (car b) mpg))
           (indices 'out 0 1)
           (indices 'a 0 1)
           (indices 'b 0 1))))


(disable tv-alu-help$value-base-case)
```

```
(prove-lemma f$tv-alu-help-v-threefix-mpg (rewrite)
  (implies
   (equal (length mpg) 7)
   (equal (f$tv-alu-help c a b (v-threefix mpg) tree)
          (f$tv-alu-help c a b mpg tree)))
  ;;Hint
  ((induct (f$tv-alu-help c a b mpg tree))
   (expand (f$tv-alu-help c a b mpg tree)
           (f$tv-alu-help c a b (v-threefix mpg) tree))
   (disable-theory f-gates)))


(prove-lemma tv-alu-help$value (rewrite)
  (implies
   (and (tv-alu-help& netlist tree)
        (properp a) (equal (length a) (tree-size tree))
        (properp b) (equal (length b) (tree-size tree))
        (properp mpg) (equal (length mpg) 7))
   (equal (dual-eval 0 (index 'tv-alu-help (tree-number tree))
                     (cons c (append a (append b mpg)))
                     state netlist)
          (f$tv-alu-help c a b mpg tree)))
  ;;Hint
  ((induct (tv-alu-help-induction tree c a b mpg state netlist))
   (enable tv-alu-help-lemma-crock t-carry$value b-and$value v-buf$value
           tv-alu-help$value-base-case tv-alu-help*$destructure)
   (disable indices *1*indices open-indices f-and tree-number
            t-carry-p-g-carry)
   (expand (tv-alu-help& netlist tree)
           (f$tv-alu-help c a b mpg tree))))


(disable tv-alu-help$value)
```

14.52 "post-alu.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    POST-ALU.EVENTS
;;;
;;;   CARRY and OVERFLOW help, and TV-SHIFT-OR-BUF.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  CARRY-OUT-HELP


(defn carry-out-help (a0 result zero op0 op1 op2 op3)
  (let ((result- (b-not result))
        (zero-   (b-not zero))
        (op0-    (b-not op0))
        (op1-    (b-not op1))
        (op2-    (b-not op2))
        (op3-    (b-not op3)))
    (let ((op0    (b-not op0-))
          (op1    (b-not op1-))
          (op2    (b-not op2-))
          (op3    (b-not op3-)))

      (b-and (b-nand3 (b-nand4 op3- (b-nand op0- op1-) op2- result)
                      (b-nand3 op3- op2 result-)
                      (b-nand4 op3 op2- (b-nand op0 op1) a0))
             zero-))))


(defn-to-module carry-out-help)


(prove-lemma carry-out-help-congruence (rewrite)
  (implies
   x
   (equal (equal (carry-out-help x a zero op0 op1 op2 op3)
                 (carry-out-help t a zero op0 op1 op2 op3))
          t)))
```

```
(prove-lemma carry-out-help$value-zero (rewrite)
  (implies
   (carry-out-help& netlist)
   (equal (dual-eval 0 'carry-out-help
                     (list a0 result t op0 op1 op2 op3)
                     state netlist)
          (list f)))
  ;;Hint
  ((enable carry-out-help& b-not$value b-and$value b-not f-and-rewrite)
   (disable-theory f-gates b-gates)))


(disable carry-out-help$value-zero)

;;;   OVERFLOW-HELP
;;;
;;;   This logic is optimized for speed on the basis that RN will arrive last.


(defn overflow-help (rn an bn zero op0 op1 op2 op3)
  (let ((an-   (b-not an))
        (zero- (b-not zero))
        (op1-  (b-not op1))
        (op2-  (b-not op2))
        (op3-  (b-not op3)))
    (let ((an   (b-not an-))
          (zero (b-not zero-))
          (op2  (b-not op2-)))
      (b-if rn
            (b-nor (b-nand (b-nor (b-nand3 op3-
                                           (b-or3 op1- op2- (b-xor an bn))
                                           (b-nand3 op1- op2 an-))
                                  (b-nand (b-nand3 op1 op2- (b-xor an bn))
                                          (b-nand3 op1- op2- an)))
                           zero-)
                   (b-nand3 (b-nand op2 an-)
                            (b-nand3 op0 op1- an)
                            (b-nand op2- an)))
            (b-nor (b-nand (b-nor (b-nand3 op3-
                                           (b-or3 op1- op2- (b-xor an bn))
                                           (b-nand3 op1- op2 an-))
                                  (b-nand (b-nand3 op1 op2- (b-xor an bn))
                                          (b-nand3 op1- op2- an)))
                           zero-)
                   (b-not (b-nand3 (b-nand op2 an-)
                                   (b-nand3 op0 op1- an)
                                   (b-nand op2- an)))))

      )))


(defn-to-module overflow-help)
```

```
(prove-lemma overflow-help$value-zero (rewrite)
  (implies
   (and (overflow-help& netlist)
        (boolp rn))
   (equal (dual-eval 0 'overflow-help
                     (list rn an bn t op0 op1 op2 op3)
                     state netlist)
          (list f)))
  ;;Hint
  ((enable overflow-help& b-nand$value b-not$value b-nor$value f-nand
           f-nor b-if$value b-not f-and3 f-and-rewrite boolp b-if)
   (disable-theory f-gates b-gates)))


(disable overflow-help$value-zero)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   THE SHIFT UNIT
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;  V-SHIFT-RIGHT-NAMES  -- Like V-SHIFT-RIGHT, but doesn't BOOLFIX the
;;;  vector.  Used to create a "shifted" list of names to be used as the
;;;  shifted input to the shift mutiplexor.


(defn v-shift-right-names (a si)
  (if (nlistp a)
      nil
    (append (cdr a) (list si))))


(prove-lemma length-v-shift-right-names (rewrite)
  (equal (length (v-shift-right-names a si))
         (length a))
  ;;Hint
  ((enable length)))


(prove-lemma properp-shift-right-names (rewrite)
  (properp (v-shift-right-names a si))
  ;;Hint
  ((enable properp)))


(prove-lemma collect-value-v-shift-right (rewrite)
  (equal (v-threefix (collect-value (v-shift-right-names a si) alist))
         (fv-shift-right (collect-value a alist) (value si alist)))
  ;;Hint
  ((enable collect-value append fv-shift-right v-threefix)))

;;;   SHIFT-OR-BUF-CNTL
```

```
(defn shift-or-buf-cntl (c an zero op0 op1 op2 op3)
  (let ((op0- (b-not op0))
        (op1- (b-not op1))
        (op2- (b-not op2)))
    (let ((decode-ror (b-and op0- op1-))
          (decode-asr op0))
      (let ((ror-si (b-and decode-ror c))
            (asr-si (b-and decode-asr an)))
        (let ((si (b-or asr-si ror-si))
              (t1 (b-nand op2- op3))
              (t2 (b-and op0 op1)))
          (list (b-or3 t2 t1 zero)
                si))))))


(defn-to-module shift-or-buf-cntl)


(prove-lemma shift-or-buf-cntl$value-zero (rewrite)
  (implies
   (shift-or-buf-cntl& netlist)
   (equal (car (dual-eval 0 'shift-or-buf-cntl
                          (list c an t op0 op1 op2 op3)
                          state netlist))
          t))
   ;;Hint
  ((enable shift-or-buf-cntl& b-or3$value f-or3 f-or-rewrite)
   (disable-theory f-gates b-gates)))


(defn f$shift-or-buf (c a an zero op0 op1 op2 op3)
  (let ((pass (car (f$shift-or-buf-cntl c an zero op0 op1 op2 op3)))
        (si   (cadr (f$shift-or-buf-cntl c an zero op0 op1 op2 op3))))
    (fv-if pass a (fv-shift-right a si))))


(disable f$shift-or-buf)


(defn shift-or-buf (c a an zero op0 op1 op2 op3)
  (let ((pass (car (shift-or-buf-cntl c an zero op0 op1 op2 op3)))
        (si   (cadr (shift-or-buf-cntl c an zero op0 op1 op2 op3))))
    (v-if pass a (v-shift-right a si))))


(disable shift-or-buf)
```

```
(prove-lemma properp-length-f$shift-or-buf (rewrite)
  (and (properp (f$shift-or-buf c a an zero op0 op1 op2 op3))
       (equal (length (f$shift-or-buf c a an zero op0 op1 op2 op3))
              (length a)))
  ;;Hint
  ((enable f$shift-or-buf)))


(prove-lemma length-shift-or-buf (rewrite)
   (equal (length (shift-or-buf c a an zero op1 op2 op3 op4))
          (length a))
   ;;Hint
   ((expand (shift-or-buf c a an zero op1 op2 op3 op4))))


(prove-lemma bvp-shift-or-buf (rewrite)
  (bvp (shift-or-buf c a an zero op1 op2 op3 op4))
  ;;Hint
  ((expand (shift-or-buf c a an zero op1 op2 op3 op4))))


(prove-lemma f$shift-or-buf=shift-or-buf (rewrite)
  (implies
   (and (boolp c)
        (boolp an)
        (boolp zero)
        (boolp op1)
        (boolp op2)
        (boolp op3)
        (boolp op4)
        (bvp a))
   (equal (f$shift-or-buf c a an zero op1 op2 op3 op4)
          (shift-or-buf c a an zero op1 op2 op3 op4)))
  ;;Hint
  ((enable f$shift-or-buf shift-or-buf)
   (disable-theory f-gates)))

;;;  Lemmas about the shift unit.


(prove-lemma shift-or-buf-is-buf (rewrite)
  (implies (and (bvp a)
                (or (and op0 op1)
                    op2
                    (not op3)
                    zero))
           (equal (shift-or-buf c a an zero op0 op1 op2 op3)
                  a))
  ;;Hint
  ((enable shift-or-buf)
   (disable v-shift-right)))
```

```
(prove-lemma shift-or-buf-is-asr (rewrite)
  (implies (and (bvp a)
                (equal an (nth (sub1 (length a)) a))
                op0
                (not op1)
                (not op2)
                op3
                (not zero))
           (equal (shift-or-buf c a an zero op0 op1 op2 op3)
                  (v-asr a)))
  ;;Hint
  ((enable shift-or-buf v-shift-right)))


(prove-lemma shift-or-buf-is-ror (rewrite)
  (implies (and (bvp a)
                (boolp c)
                (not op0)
                (not op1)
                (not op2)
                op3
                (not zero))
           (equal (shift-or-buf c a an zero op0 op1 op2 op3)
                  (v-ror a c)))
  ;;Hint
  ((enable shift-or-buf v-shift-right)))


(prove-lemma shift-or-buf-is-lsr (rewrite)
  (implies (and (bvp a)
                (not op0)
                op1
                (not op2)
                op3
                (not zero))
           (equal (shift-or-buf c a an zero op0 op1 op2 op3)
                  (v-lsr a)))
  ;;Hint
  ((enable shift-or-buf)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    TV-SHIFT-OR-BUF*
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

;;;  TV-SHIFT-OR-BUF* -- Hardware implementation of SHIFT-OR-BUF, using TV-IF
;;;  for the selector.
```

```
(defn tv-shift-or-buf* (tree)
  (let ((a-names (indices 'a 0 (tree-size tree)))
        (out-names (indices 'out 0 (tree-size tree))))
  (list
   ;;  Name
   (index 'tv-shift-or-buf (tree-number tree))
   ;;  Inputs
   (cons 'c (append a-names '(an zero op0 op1 op2 op3)))
   ;;  Outputs
   out-names
   ;;  Body
   (list
    '(cntl (ctl si) shift-or-buf-cntl (c an zero op0 op1 op2 op3))
    (list 'mux
          out-names
          (index 'tv-if (tree-number tree))
          (cons 'ctl (append a-names
                             (v-shift-right-names a-names 'si)))))
  ;;  States
  nil)))


(destructuring-lemma tv-shift-or-buf*)


(defn tv-shift-or-buf& (netlist tree)
  (and (equal (lookup-module (index 'tv-shift-or-buf (tree-number tree))
                             netlist)
              (tv-shift-or-buf* tree))

       (shift-or-buf-cntl& (delete-module
                            (index 'tv-shift-or-buf (tree-number tree))
                            netlist))
       (tv-if& (delete-module (index 'tv-shift-or-buf (tree-number tree))
                              netlist)
               tree)))


(disable tv-shift-or-buf&)


(defn tv-shift-or-buf$netlist (tree)
  (cons (tv-shift-or-buf* tree)
        (union (tv-if$netlist tree)
               (shift-or-buf-cntl$netlist))))
```

```
(prove-lemma tv-shift-or-buf$value (rewrite)
  (implies
   (and (tv-shift-or-buf& netlist tree)
        (equal (length a) (tree-size tree))
        (properp a))
   (equal (dual-eval 0 (index 'tv-shift-or-buf (tree-number tree))
                     (cons c (append a (list an zero op0 op1 op2 op3)))
                     state netlist)
          (f$shift-or-buf c a an zero op0 op1 op2 op3)))
  ;;Hint
  ((enable f$shift-or-buf tv-shift-or-buf&
           shift-or-buf-cntl$value tv-if$value
           tv-shift-or-buf*$destructure fv-if-rewrite)
   (disable-theory f-gates)
   (disable fv-shift-right v-shift-right-names)))


(disable tv-shift-or-buf$value)


(prove-lemma tv-shift-or-buf$value-zero (rewrite)
  (implies
   (and (tv-shift-or-buf& netlist tree)
        (equal (length a) (tree-size tree))
        (bvp a))
   (equal (dual-eval 0 (index 'tv-shift-or-buf (tree-number tree))
                     (cons c (append a (list an t op0 op1 op2 op3)))
                     state netlist)
          a))
  ;;Hint
  ((enable tv-shift-or-buf&
           tv-if$value tv-shift-or-buf*$destructure
           shift-or-buf-cntl$value-zero)
   (disable-theory f-gates b-gates)
   (disable v-shift-right v-shift-right-names)))


(disable tv-shift-or-buf$value-zero)
```

14.53 "core-alu.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;   CORE-ALU.EVENTS
;;;
;;;   CORE-ALU is the hardware implementation of the high-level specification
;;;   V-ALU.  In addition to the functions provided by V-ALU, CORE-ALU also can
;;;   be forced to 0, and includes INC B and DEC B operations for address
;;;   calculations.  Note that the control lines for the function generators
;;;   (MPG) are supplied by the control logic.
;;;
;;;   Since CORE-ALU is defined in terms of F$FAST-ZERO, only ALU sizes of >= 3
;;;   bits have the desired properties.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn f$core-alu (c a b zero mpg op tree)
  (let ((op0 (car op))
        (op1 (cadr op))
        (op2 (caddr op))
        (op3 (cadddr op)))
    (let ((last-bit (sub1 (length a))))
      (let ((alu-help (f$tv-alu-help c a b mpg tree)))
        (let ((alu-p   (car alu-help))
              (alu-g   (cadr alu-help))
              (alu-sum (cddr alu-help)))
          (let ((alu-carry (f$t-carry c alu-p alu-g))
                (out (f$shift-or-buf c alu-sum (nth (sub1 (length a)) a)
                                     zero op0 op1 op2 op3)))
            (cons (f$carry-out-help (nth 0 a) alu-carry zero op0 op1 op2 op3)
                  (cons (f$overflow-help (nth last-bit alu-sum)
                                         (nth last-bit a)
                                         (nth last-bit b)
                                         zero op0 op1 op2 op3)
                        (cons (f$fast-zero out)
                              out)))))))))


(disable f$core-alu)
```

```
(defn core-alu (c a b zero mpg op tree)
  (let ((op0 (car op))
        (op1 (cadr op))
        (op2 (caddr op))
        (op3 (cadddr op)))
    (let ((last-bit (sub1 (length a))))
      (let ((alu-help (tv-alu-help c a b mpg tree)))
        (let ((alu-p   (car alu-help))
              (alu-g   (cadr alu-help))
              (alu-sum (cddr alu-help)))
          (let ((alu-carry (t-carry c alu-p alu-g))
                (out (shift-or-buf c alu-sum (nth (sub1 (length a)) a)
                                   zero op0 op1 op2 op3)))
            (cons (carry-out-help (nth 0 a) alu-carry zero op0 op1 op2 op3)
                  (cons (overflow-help (nth last-bit alu-sum)
                                       (nth last-bit a)
                                       (nth last-bit b)
                                       zero op0 op1 op2 op3)
                        (cons (v-zerop out)
                              out)))))))))

(disable core-alu)

(prove-lemma properp-length-f$core-alu (rewrite)
  (and (properp (f$core-alu c a b zero mpg op tree))
       (equal (length (f$core-alu c a b zero mpg op tree))
              (add1 (add1 (add1 (tree-size tree))))))
  ;;Hint
  ((enable f$core-alu)))

(prove-lemma f$core-alu=core-alu (rewrite)
  (implies
   (and (boolp c)
        (bvp a) (equal (length a) (tree-size tree))
        (bvp b) (equal (length b) (tree-size tree))
        (geq (length a) 3)
        (boolp zero)
        (bvp mpg) (equal (length mpg) 7)
        (bvp op) (equal (length op) 4))
   (equal (f$core-alu c a b zero mpg op tree)
          (core-alu c a b zero mpg op tree)))
  ;;Hint
  ((enable core-alu f$core-alu bvp-length)))

(prove-lemma length-core-alu (rewrite)
  (equal (length (core-alu c a b zero mpg op tree))
         (add1 (add1 (add1 (tree-size tree)))))
  ;;Hint
  ((enable core-alu)))
```

```
(prove-lemma boolp-bvp-core-alu (rewrite)
  (and (boolp (c (core-alu c a b zero mpg op tree)))
       (boolp (v (core-alu c a b zero mpg op tree)))
       (boolp (zb (core-alu c a b zero mpg op tree)))
       (bvp (bv (core-alu c a b zero mpg op tree)))
       (bvp (core-alu c a b zero mpg op tree)))
  ;;Hint
  ((enable core-alu bvp c v zb bv)
   (disable carry-in-help carry-out-help overflow-help shift-or-buf
            v-zerop)))


(prove-lemma core-alu-works-for-all-normal-cases$crock (rewrite)
  (implies
   (and (bvp a)
        (equal (length a) (tree-size tree))
        (nth 0 a))
   (equal (nth 0 a) t))
  ;;hint
  ((enable open-nth bvp)))


(disable core-alu-works-for-all-normal-cases$crock)
```

```
(prove-lemma core-alu-works-for-all-normal-cases (rewrite)
  (and

   (implies
    (and (bv2p a b)
         (equal (length a) (tree-size tree))
         (boolp c))
    (equal (v-alu c a b #v0000)
           (core-alu (carry-in-help (cons c (cons f #v0000)))
                     a b f (mpg #v000000) #v0000 tree)))

   (implies
    (and (bv2p a b)
         (equal (length a) (tree-size tree))
         (boolp c))
    (equal (v-alu c a b #v0001)
           (core-alu (carry-in-help (cons c (cons f #v0001)))
                     a b f (mpg #v000100) #v0001 tree)))

   (implies
    (and (bv2p a b)
         (equal (length a) (tree-size tree))
         (boolp c))
    (equal (v-alu c a b #v0010)
           (core-alu (carry-in-help (cons c (cons f #v0010)))
                     a b f (mpg #v001000) #v0010 tree)))

   (implies
    (and (bv2p a b)
         (equal (length a) (tree-size tree))
         (boolp c))
    (equal (v-alu c a b #v0011)
           (core-alu (carry-in-help (cons c (cons f #v0011)))
                     a b f (mpg #v001100) #v0011 tree)))

   (implies
    (and (bv2p a b)
         (equal (length a) (tree-size tree))
         (boolp c))
    (equal (v-alu c a b #v0100)
           (core-alu (carry-in-help (cons c (cons f #v0100)))
                     a b f (mpg #v010000) #v0100 tree)))

   (implies
    (and (bv2p a b)
         (equal (length a) (tree-size tree))
         (boolp c))
    (equal (v-alu c a b #v0101)
           (core-alu (carry-in-help (cons c (cons f #v0101)))
                     a b f (mpg #v010100) #v0101 tree)))

   (implies
    (and (bv2p a b)
         (equal (length a) (tree-size tree))
```

```
        (boolp c))
    (equal (v-alu c a b #v0110)
           (core-alu (carry-in-help (cons c (cons f #v0110)))
                     a b f (mpg #v011000) #v0110 tree)))

   (implies
    (and (bv2p a b)
         (equal (length a) (tree-size tree))
         (boolp c))
    (equal (v-alu c a b #v0111)
           (core-alu (carry-in-help (cons c (cons f #v0111)))
                     a b f (mpg #v011100) #v0111 tree)))

   (implies
    (and (bv2p a b)
         (equal (length a) (tree-size tree))
         (boolp c))
    (equal (v-alu c a b #v1000)
           (core-alu (carry-in-help (cons c (cons f #v1000)))
                     a b f (mpg #v100000) #v1000 tree)))

   (implies
    (and (bv2p a b)
         (equal (length a) (tree-size tree))
         (boolp c))
    (equal (v-alu c a b #v1001)
           (core-alu (carry-in-help (cons c (cons f #v1001)))
                     a b f (mpg #v100100) #v1001 tree)))

   (implies
    (and (bv2p a b)
         (equal (length a) (tree-size tree))
         (boolp c))
    (equal (v-alu c a b #v1010)
           (core-alu (carry-in-help (cons c (cons f #v1010)))
                     a b f (mpg #v101000) #v1010 tree)))

;; The one below breaks with the new CARRY-OUT-HELP.
   (implies
    (and (bv2p a b)
         (equal (length a) (tree-size tree))
         (boolp c))
    (equal (v-alu c a b #v1011)
           (core-alu (carry-in-help (cons c (cons f #v1011)))
                     a b f (mpg #v101100) #v1011 tree)))

   (implies
    (and (bv2p a b)
         (equal (length a) (tree-size tree))
         (boolp c))
    (equal (v-alu c a b #v1100)
           (core-alu (carry-in-help (cons c (cons f #v1100)))
                     a b f (mpg #v110000) #v1100 tree)))

   (implies
```

```
    (and (bv2p a b)
         (equal (length a) (tree-size tree))
         (boolp c))
    (equal (v-alu c a b #v1101)
           (core-alu (carry-in-help (cons c (cons f #v1101)))
                     a b f (mpg #v110100) #v1101 tree)))

   (implies
    (and (bv2p a b)
         (equal (length a) (tree-size tree))
         (boolp c))
    (equal (v-alu c a b #v1110)
           (core-alu (carry-in-help (cons c (cons f #v1110)))
                     a b f (mpg #v111000) #v1110 tree)))

   (implies
    (and (bv2p a b)
         (equal (length a) (tree-size tree))
         (boolp c))
    (equal (v-alu c a b #v1111)
           (core-alu (carry-in-help (cons c (cons f #v1111)))
                     a b f (mpg #v111100) #v1111 tree))))
  ;;Hint
  ((disable t-carry tv-alu-help)
   (enable
    core-alu
    v-alu
    boolp
    mpg
    carry-in-help
    tv-alu-help-tv-neg-works
    tv-alu-help-v-and-works
    tv-alu-help-v-or-works
    tv-alu-help-v-xor-works
    tv-alu-help-v-not-works
    tv-alu-help-v-buf-works
    tv-alu-help-tv-adder-works
    tv-alu-help-tv-subtracter-works
    tv-alu-help-tv-inc-a-works
    tv-alu-help-tv-dec-a-works
    tv-alu-help-tv-neg-works
    tv-adder-as-p-g-sum
    core-alu-works-for-all-normal-cases$crock)))


(disable core-alu-works-for-all-normal-cases)
```

```
(prove-lemma cases-on-a-4-bit-bvp ()
  (implies
   (and (bvp op)
        (equal (length op) 4)
        (cond ((equal op #v0000) p)
              ((equal op #v0001) p)
              ((equal op #v0010) p)
              ((equal op #v0011) p)
              ((equal op #v0100) p)
              ((equal op #v0101) p)
              ((equal op #v0110) p)
              ((equal op #v0111) p)
              ((equal op #v1000) p)
              ((equal op #v1001) p)
              ((equal op #v1010) p)
              ((equal op #v1011) p)
              ((equal op #v1100) p)
              ((equal op #v1101) p)
              ((equal op #v1110) p)
              ((equal op #v1111) p)
              (t t)))
   p)
  ;;Hint
  ((enable equal-length-add1)))


(prove-lemma core-alu-is-v-alu (rewrite)
  (implies
   (and (bv2p a b)
        (equal (length a) (tree-size tree))
        (bvp op)
        (equal (length op) 4)
        (not zero)
        (equal mpg (mpg (cons zero (cons f op))))
        (boolp c))
   (equal (core-alu (carry-in-help (cons c (cons f op)))
                    a b zero mpg op tree)
          (v-alu c a b op)))
  ;;Hint
  ((use (cases-on-a-4-bit-bvp
         (p (equal (core-alu (carry-in-help (cons c (cons f op)))
                             a b zero mpg op tree)
                   (v-alu c a b op)))))
   (disable boolp-lemmas bv2p bvp length core-alu v-alu tree-size
            mpg *1*mpg)
   (enable core-alu-works-for-all-normal-cases)))
```

```
(prove-lemma core-alu-works-for-zero-case (rewrite)
  (implies
   (and (equal (length a) (tree-size tree))
        zero)
   (equal (core-alu t a b zero #v1000000 op tree)
          (cvzbv f f (make-list (length a) f))))
  ;;Hint
  ((enable core-alu tv-alu-help-zero)))


(prove-lemma core-alu-works-as-inc-b (rewrite)
  (implies
   (and (bv2p a b)
        (equal (length a) (tree-size tree))
        (not zero)
        swap)
   (equal (bv (core-alu (carry-in-help (cons c (cons zero (alu-inc-op))))
                        a b zero
                        (mpg (cons zero (cons swap (alu-inc-op))))
                        (alu-inc-op) tree))
          (v-inc b)))
  ;;Hint
  ((disable tv-alu-help)
   (enable alu-inc-op v-inc bv core-alu mpg carry-in-help
           tv-alu-help-tv-inc-b-works tv-adder-as-p-g-sum)))


(prove-lemma core-alu-works-as-dec-b (rewrite)
  (implies
   (and (bv2p a b)
        (equal (length a) (tree-size tree))
        (not zero)
        swap)
   (equal (bv (core-alu (carry-in-help (cons c (cons zero (alu-dec-op))))
                        a b zero
                        (mpg (cons zero (cons swap (alu-dec-op))))
                        (alu-dec-op) tree))
          (v-dec b)))
  ;;Hint
  ((disable tv-alu-help)
   (enable alu-dec-op v-dec bv core-alu mpg carry-in-help
           tv-alu-help-tv-dec-b-works tv-adder-as-p-g-sum)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    CORE-ALU*
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defn core-alu* (tree)
  (let ((a-names       (indices 'a 0 (tree-size tree)))
        (b-names       (indices 'b 0 (tree-size tree)))
        (mpg-names     (indices 'mpg 0 7))
        (op-names      (indices 'op 0 4))
        (alu-out-names (indices 'alu-out 0 (tree-size tree)))
        (out-names     (indices 'out 0 (tree-size tree)))
        (last-a        (index 'a (sub1 (tree-size tree))))
        (last-b        (index 'b (sub1 (tree-size tree))))
        (last-alu-out  (index 'alu-out (sub1 (tree-size tree)))))
    (list
     ;;  Name
     (index 'core-alu (tree-number tree))
     ;;  Inputs
     (cons 'c (append a-names (append b-names
                                       (cons 'zero
                                             (append mpg-names op-names)))))
     ;;  Outputs
     (cons 'carry (cons 'overflow (cons 'zerop out-names)))
     ;;  Body
     (list
      (list 'm-alu
            (cons 'p (cons 'g alu-out-names))
            (index 'tv-alu-help (tree-number tree))
            (cons 'c (append a-names (append b-names mpg-names))))
      (list 'm-alu-carry
            '(alu-carry)
            't-carry
            '(c p g))
      (list 'm-carry-out-help
            '(carry)
            'carry-out-help
            (cons (index 'a 0)
                  (list 'alu-carry 'zero
                        #i(op 0) #i(op 1) #i(op 2) #i(op 3))))
      (list 'm-overflow-help
            '(overflow)
            'overflow-help
            (list last-alu-out last-a last-b 'zero
                  #i(op 0) #i(op 1) #i(op 2) #i(op 3)))
      (list 'm-shift
            out-names
            (index 'tv-shift-or-buf (tree-number tree))
            (cons 'c (append alu-out-names
                             (list last-a 'zero
                                   #i(op 0) #i(op 1)
                                   #i(op 2) #i(op 3)))))
      (list 'm-zerop
            '(zerop)
            #i(fast-zero (tree-size tree))
            out-names))
     ;;  States
     nil)))
```

```
(destructuring-lemma core-alu*)


(defn core-alu& (netlist tree)
  (and
   (equal (lookup-module (index 'core-alu (tree-number tree)) netlist)
          (core-alu* tree))
   (let ((netlist
          (delete-module (index 'core-alu (tree-number tree)) netlist)))
     (and (tv-alu-help&     netlist tree)
          (t-carry&         netlist)
          (carry-out-help&  netlist)
          (overflow-help&   netlist)
          (tv-shift-or-buf& netlist tree)
          (fast-zero&       netlist (tree-size tree))))))


(disable core-alu&)


(defn core-alu$netlist (tree)
  (cons
   (core-alu* tree)
   (union
    (tv-alu-help$netlist tree)
    (union (t-carry$netlist)
           (union (carry-out-help$netlist)
                  (union (overflow-help$netlist)
                         (union (tv-shift-or-buf$netlist tree)
                                (fast-zero$netlist (tree-size tree)))))))))


(prove-lemma core-alu$value (rewrite)
  (implies
   (and (core-alu& netlist tree)
        (equal (length a) (tree-size tree))
        (equal (length b) (tree-size tree))
        (geq (length a) 3)
        (properp a) (properp b)
        (properp op) (equal (length op) 4)
        (properp mpg) (equal (length mpg) 7))
   (equal (dual-eval 0 (index 'core-alu (tree-number tree))
                     (cons c (append a (append b (cons zero (append mpg op)))))
                     state netlist)
          (f$core-alu c a b zero mpg op tree)))
  ;;Hint
  ((enable f$core-alu core-alu& tv-shift-or-buf$value
           core-alu*$destructure
           tv-alu-help$value t-carry$value
           carry-out-help$value overflow-help$value fast-zero$value
           open-nth)
   (disable *1*indices open-indices indices)))
```

```
(disable core-alu$value)
```

14.54 "fm9001-spec.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    FM9001-SPEC.EVENTS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#|
   FM9001 -- a processor for a new decade.

   Instruction format:
                                                         { N/A  mode-a rn-a
                                                         {  3     2      4
unused op-code store-cc set-flags mode-b rn-b a-immediate |
  4       4       4        4        2      4       1     {    a-immediate
                                                         {         9


The A operand is a 10 bit field.  If the high order bit is set, the low order
9 bits are treated as a signed immediate.  Otherwise, the low order six bits of
the A operand are a mode/register pair identical to the B operand.
```

```
(Note:  We use "a" for "rn-a" and "b" for "rn-b" below.)

Interpretation of the OP-CODE.

 0000  b <- a            Move
 0001  b <- a + 1        Increment
 0010  b <- a + b + c    Add with carry
 0011  b <- b + a        Add
 0100  b <- 0 - a        Negation
 0101  b <- a - 1        Decrement
 0110  b <- b - a - c    Subtract with borrow
 0111  b <- b - a        Subtract
 1000  b <- a >> 1       Rotate right, shifted through carry
 1001  b <- a >> 1       Arithmetic shift right, top bit copied
 1010  b <- a >> 1       Logical shift right, top bit zero
 1011  b <- b XOR a      Exclusive or
 1100  b <- b | a        Or
 1101  b <- b & a        And
 1110  b <- ~a           Not
 1111  b <- a            Move


Interpretation of the STORE-CC field.

 0000  (~c)                          Carry clear
 0001  (c)                           Carry set
 0010  (~v)                          Overflow clear
 0011  (v)                           Overflow set
 0100  (~n)                          Plus
 0101  (n)                           Negative
 0110  (~z)                          Not equal
 0111  (z)                           Equal
 1000  (~c & ~z)                     High
 1001  (c | z)                       Low or same
 1010  (n & v | ~n & ~v)             Greater or equal
 1011  (n & ~v | ~n & v)             Less than
 1100  (n & v & ~z | ~n & ~v & ~z)   Greater than
 1101  (z | n & ~v | ~n & v)         Less or equal
 1110  (t)                           True
 1111  (nil)                         False


Flags are set conditionally based on the SET-FLAGS field.

 0000  ----
 0001  ---Z
 0010  --N-
 0011  --NZ
 0100  -V--
 0101  -V-Z
 0110  -VN-
 0111  -VNZ
 1000  C---
 1001  C--Z
```

```
 1010  C-N-
 1011  C-NZ
 1100  CV--
 1101  CV-Z
 1110  CVN-
 1111  CVNZ

Addressing Modes for "a" and "b".

 00  Register Direct
 01  Register Indirect
 10  Register Indirect with Pre-decrement
 11  Register Indirect with Post-increment


Register Numbers for "a" and "b".

 0000  Register 0
 0001  Register 1
 0010  Register 2
 0011  Register 3
 0100  Register 4
 0101  Register 5
 0110  Register 6
 0111  Register 7
 1000  Register 8
 1001  Register 9
 1010  Register 10
 1011  Register 11
 1100  Register 12
 1101  Register 13
 1110  Register 14
 1111  Register 15

|#

;;;  Instruction accessors.


(defn reg-size () 4)


(defn a-immediate-p (instruction)
  (nth 9 instruction))


(defn a-immediate (instruction)
  (subrange instruction 0 8))


(defn rn-a (instruction)
  (subrange instruction 0 3))
```

```
(defn mode-a (instruction)
  (subrange instruction 4 5))


(defn rn-b (instruction)
  (subrange instruction 10 13))


(defn mode-b (instruction)
  (subrange instruction 14 15))


(defn set-flags (instruction)
  (subrange instruction 16 19))


(defn store-cc (instruction)
  (subrange instruction 20 23))


(defn op-code (instruction)
  (subrange instruction 24 27))


(disable a-immediate-p)

(disable a-immediate)

(disable rn-a)

(disable rn-b)

(disable mode-a)

(disable mode-b)

(disable set-flags)

(disable store-cc)

(disable op-code)


(deftheory ir-fields-theory
  (a-immediate-p a-immediate rn-a mode-a rn-b mode-b
                 set-flags store-cc op-code))

;;;  SET-FLAGS fields


(defn z-set (set-flags)
  (nth 0 set-flags))
```

```
(defn n-set (set-flags)
  (nth 1 set-flags))

(defn v-set (set-flags)
  (nth 2 set-flags))

(defn c-set (set-flags)
  (nth 3 set-flags))

(disable z-set)

(disable n-set)

(disable v-set)

(disable c-set)

(deftheory set-flags-theory (z-set n-set v-set c-set))

;;;  Flags fields

(defn z-flag (flags)
  (nth 0 flags))

(defn n-flag (flags)
  (nth 1 flags))

(defn v-flag (flags)
  (nth 2 flags))

(defn c-flag (flags)
  (nth 3 flags))

(disable z-flag)

(disable n-flag)

(disable v-flag)

(disable c-flag)
```

```
(deftheory flags-theory (z-flag n-flag v-flag c-flag))

;;;  Interpretations of accessors.


(defn reg-direct-p (mode)
  (equal mode #v00))


(defn reg-indirect-p (mode)
  (equal mode #v01))


(defn pre-dec-p (mode)
  (equal mode #v10))


(defn post-inc-p (mode)
  (equal mode #v11))


(disable reg-direct-p)

(disable reg-indirect-p)

(disable pre-dec-p)

(disable post-inc-p)


(deftheory reg-mode-theory
  (reg-direct-p reg-indirect-p pre-dec-p post-inc-p))

;;;  Interpretation of STORE-CC
```

```
(defn store-resultp (store-cc flags)
  (let ((c (c-flag flags))
        (v (v-flag flags))
        (n (n-flag flags))
        (z (z-flag flags)))
    (let ((c~ (not c))
          (v~ (not v))
          (n~ (not n))
          (z~ (not z)))

      (cond ((equal store-cc #v0000) c~)
            ((equal store-cc #v0001) c)
            ((equal store-cc #v0010) v~)
            ((equal store-cc #v0011) v)
            ((equal store-cc #v0100) n~)
            ((equal store-cc #v0101) n)
            ((equal store-cc #v0110) z~)
            ((equal store-cc #v0111) z)
            ((equal store-cc #v1000) (and c~ z~))
            ((equal store-cc #v1001) (or c z))
            ((equal store-cc #v1010) (or (and n v) (and n~ v~)))
            ((equal store-cc #v1011) (or (and n v~) (and n~ v)))
            ((equal store-cc #v1100) (or (and n v z~) (and n~ v~ z~)))
            ((equal store-cc #v1101) (or z (and n v~) (and n~ v)))
            ((equal store-cc #v1110) t)
            (t                       f)))))


(disable store-resultp)

;;;  UPDATE-FLAGS set-flags cvzbv


(defn update-flags (flags set-flags cvzbv)
  (list (b-if (z-set set-flags) (zb cvzbv) (z-flag flags))
        (b-if (n-set set-flags) (n  cvzbv) (n-flag flags))
        (b-if (v-set set-flags) (v  cvzbv) (v-flag flags))
        (b-if (c-set set-flags) (c  cvzbv) (c-flag flags))))


(disable update-flags)

;;;  An FM9001 state is a list of two items: The processor state, which
;;;  consists of the register file and flags, and the memory state.


(defn regs  (state) (nth 0 state))

(defn flags (state) (nth 1 state))


(disable regs)
```

```
(disable flags)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  The FM9001 instruction interpreter.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; FM9001-ALU-OPERATION -- Computes, and conditionally stores the result.


(defn fm9001-alu-operation (regs flags mem ins operand-a operand-b b-address)
  (let ((op-code   (op-code ins))
        (store-cc  (store-cc ins))
        (set-flags (set-flags ins))
        (mode-b    (mode-b   ins))
        (rn-b      (rn-b    ins)))
    (let ((cvzbv  (v-alu (c-flag flags) operand-a operand-b op-code))
          (storep (store-resultp store-cc flags)))
      (let ((bv (bv cvzbv)))
        (let ((new-regs   (if (and storep (reg-direct-p mode-b))
                              (write-mem rn-b regs bv)
                            regs))
              (new-flags  (update-flags flags set-flags cvzbv))
              (new-mem    (if (and storep (not (reg-direct-p mode-b)))
                              (write-mem b-address mem bv)
                            mem)))

          (list (list new-regs new-flags) new-mem))))))

;;;  FM9001-OPERAND-B -- Readies the B operand, and side-effects the operand B
;;;  register.  The B-ADDRESS is held for the final stage.
```

```
(defn fm9001-operand-b (regs flags mem ins operand-a)
  (let ((mode-b (mode-b ins))
        (rn-b   (rn-b  ins)))
    (let ((reg (read-mem rn-b regs)))
      (let ((reg- (v-dec reg))
            (reg+ (v-inc reg)))
        (let ((b-address (if (pre-dec-p mode-b)
                             reg-
                           reg)))
          (let ((operand-b (if (reg-direct-p mode-b)
                               reg
                             (read-mem b-address mem)))
                (new-regs (if (pre-dec-p mode-b)
                              (write-mem rn-b regs reg-)
                            (if (post-inc-p mode-b)
                                (write-mem rn-b regs reg+)
                              regs))))

            (FM9001-alu-operation new-regs flags mem ins operand-a operand-b
                                  b-address)))))))

;;;  FM9001-OPERAND-A -- Readies the A operand, and side-effects the operand A
;;;  register.


(defn fm9001-operand-a (regs flags mem ins)
  (let ((a-immediate-p (a-immediate-p ins))
        (a-immediate   (sign-extend (a-immediate ins) 32))
        (mode-a        (mode-a ins))
        (rn-a          (rn-a  ins)))
    (let ((reg (read-mem rn-a regs)))
      (let ((reg- (v-dec reg))
            (reg+ (v-inc reg)))
        (let ((operand-a (if a-immediate-p
                             a-immediate
                           (if (reg-direct-p mode-a)
                               reg
                             (if (pre-dec-p mode-a)
                                 (read-mem reg- mem)
                               (read-mem reg mem))))))
          (let ((new-regs (if a-immediate-p
                              regs
                            (if (pre-dec-p mode-a)
                                (write-mem rn-a regs reg-)
                              (if (post-inc-p mode-a)
                                  (write-mem rn-a regs reg+)
                                regs)))))

            (FM9001-operand-b new-regs flags mem ins operand-a)))))))

;;;  FM9001-FETCH -- Fetches the instruction and increments the PC.
```

```
(defn FM9001-FETCH (regs flags mem pc-reg)
  (let ((pc (read-mem pc-reg regs)))
    (let ((ins (read-mem pc mem)))
      (let ((pc+1 (v-inc pc)))
        (let ((new-regs (write-mem pc-reg regs pc+1)))
          (FM9001-operand-a new-regs flags mem ins))))))

;;;  FM9001-STEP -- Unpacks the state.


(defn FM9001-step (state pc-reg)
  (let ((p-state (car state))
        (mem     (cadr state)))
    (FM9001-fetch (regs p-state) (flags p-state) mem pc-reg)))


(disable fm9001-step)

;;;  FM9001 -- Simulates N instructions, using register 15 as the PC.


(defn FM9001 (state n)
  (if (zerop n)
      state
    (FM9001 (FM9001-step state (nat-to-v 15 (reg-size)))
            (sub1 n))))


(disable fm9001)

;;;   FM9001-INTERPRETER
;;;
;;;   Simulates N instructions with a given PC.


(defn FM9001-interpreter (state pc-reg n)
  (if (zerop n)
      state
    (FM9001-interpreter
     (FM9001-step state pc-reg)
     pc-reg
     (sub1 n))))


(disable FM9001-interpreter)
```

```
(prove-lemma open-FM9001-interpreter (rewrite)
  (and
   (implies
    (zerop n)
    (equal (FM9001-interpreter state pc-reg n)
           state))
   (implies
    (not (zerop n))
    (equal (FM9001-interpreter state pc-reg n)
           (FM9001-interpreter
            (FM9001-step state pc-reg)
            pc-reg
            (sub1 n)))))
  ;;Hint
  ((enable fm9001-interpreter)))

;;;  FM9001-INTR -- The PC-REG-INPUT is used to determine which register is the
;;;  PC.


(defn FM9001-intr (state pc-reg-input)
  (if (nlistp pc-reg-input)
      state
      (FM9001-intr (FM9001-step state (car pc-reg-input))
                   (cdr pc-reg-input))))


(disable fm9001-intr)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    LEMMAS
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(prove-lemma properp-length-ir-accessors (rewrite)
  (and
   (properp (a-immediate i-reg))
   (equal (length (a-immediate i-reg)) 9)
   (properp (rn-a i-reg))
   (equal (length (rn-a i-reg)) 4)
   (properp (mode-a i-reg))
   (equal (length (mode-a i-reg)) 2)
   (properp (rn-b i-reg))
   (equal (length (rn-b i-reg)) 4)
   (properp (mode-b i-reg))
   (equal (length (mode-b i-reg)) 2)
   (properp (set-flags i-reg))
   (equal (length (set-flags i-reg)) 4)
   (properp (store-cc i-reg))
   (equal (length (store-cc i-reg)) 4)
   (properp (op-code i-reg))
   (equal (length (op-code i-reg)) 4))
  ;;Hint
  ((enable a-immediate rn-a rn-b mode-a mode-b set-flags store-cc op-code)))


(prove-lemma bvp-ir-accessors (rewrite)
  (implies
   (and (bvp i-reg)
        (equal (length i-reg) 32))
   (and
    (bvp (a-immediate i-reg))
    (bvp (rn-a i-reg))
    (bvp (mode-a i-reg))
    (bvp (rn-b i-reg))
    (bvp (mode-b i-reg))
    (bvp (set-flags i-reg))
    (bvp (store-cc i-reg))
    (bvp (op-code i-reg))))
  ;;Hint
  ((enable a-immediate rn-a rn-b mode-a mode-b set-flags store-cc op-code)))


(prove-lemma boolp-a-immediate-p (rewrite)
  (implies
   (and (bvp ir)
        (equal (length ir) 32))
   (boolp (a-immediate-p ir)))
  ;;Hint
  ((enable a-immediate-p)))
```

```
(prove-lemma bvp-length-update-flags (rewrite)
  (and
   (bvp (update-flags flags set-flags cvzbv))
   (equal (length (update-flags flags set-flags cvzbv))
          4))
  ;;Hint
  ((enable bvp update-flags length)))


(prove-lemma boolp-c-flag-update-flags (rewrite)
  (boolp (c-flag (update-flags flags set-flags cvzbv)))
  ;;Hint
  ((enable c-flag update-flags)))


(prove-lemma bvp-a-immediate (rewrite)
  (implies
   (and (bvp ir)
        (lessp 8 (length ir)))
   (bvp (a-immediate ir)))
  ;;Hint
  ((enable a-immediate)))


(prove-lemma reg-direct->not-reg-indirect (rewrite)
  (implies
   (reg-direct-p mode)
   (and (equal (reg-indirect-p mode) f)
        (equal (pre-dec-p mode) f)
        (equal (post-inc-p mode) f)))
  ;;Hint
  ((enable-theory reg-mode-theory)))


(prove-lemma boolp-store-result-p (rewrite)
  (implies
   (and (bvp flags)
        (equal (length flags) 4))
   (boolp (store-resultp store-cc flags)))
  ;;Hint
  ((enable store-resultp open-nth c-flag v-flag n-flag z-flag)))


(prove-lemma unary-op-code-p-op-code->v-alu=v-alu-1 (rewrite)
  (implies
   (unary-op-code-p (op-code i-reg))
   (equal (v-alu c a b (op-code i-reg))
          (v-alu-1 c a (op-code i-reg))))
  ;;Hint
  ((enable unary-op-code-p->v-alu=v-alu-1)))
```

14.55 "asm-fm9001.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    ASM-FM9001.EVENTS
;;;
;;;  A quick and dirty assembler for the FM9001.   Examples of
;;;  assember input are at the bottom of this file.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#|
   FM9001 -- a processor for a new decade.

   Instruction format:
                                                         { N/A  mode-a rn-a
                                                         {  3     2      4
unused op-code store-cc set-flags mode-b rn-b a-immediate |
  4       4       4        4        2     4        1     {    a-immediate
                                                         {         9


The A operand is a 10 bit field.  If the high order bit is set, the low order
9 bits are treated as a signed immediate.  Otherwise, the low order six bits of
the A operand are a mode/register pair identical to the B operand.
```

```
(Note:  We use "a" for "rn-a" and "b" for "rn-b" below.)

Interpretation of the OP-CODE.

 0000  b <- a            Move
 0001  b <- a + 1        Increment
 0010  b <- a + b + c    Add with carry
 0011  b <- b + a        Add
 0100  b <- 0 - a        Negation
 0101  b <- a - 1        Decrement
 0110  b <- b - a - c    Subtract with borrow
 0111  b <- b - a        Subtract
 1000  b <- a >> 1       Rotate right, shifted through carry
 1001  b <- a >> 1       Arithmetic shift right, top bit copied
 1010  b <- a >> 1       Logical shift right, top bit zero
 1011  b <- b XOR a      Exclusive or
 1100  b <- b | a        Or
 1101  b <- b & a        And
 1110  b <- ~a           Not
 1111  b <- a            Move


Interpretation of the STORE-CC field.

 0000  (~c)                         Carry clear
 0001  (c)                          Carry set
 0010  (~v)                         Overflow clear
 0011  (v)                          Overflow set
 0100  (~n)                         Plus
 0101  (n)                          Negative
 0110  (~z)                         Not equal
 0111  (z)                          Equal
 1000  (~c & ~z)                    High
 1001  (c | z)                      Low or same
 1010  (n & v | ~n & ~v)            Greater or equal
 1011  (n & ~v | ~n & v)            Less than
 1100  (n & v & ~z | ~n & ~v & ~z)  Greater than
 1101  (z | n & ~v | ~n & v)        Less or equal
 1110  (t)                          True
 1111  (nil)                        False


Flags are set conditionally based on the SET-FLAGS field.

 0000  ----
 0001  ---Z
 0010  --N-
 0011  --NZ
 0100  -V--
 0101  -V-Z
 0110  -VN-
 0111  -VNZ
 1000  C---
 1001  C--Z
```

```
 1010  C-N-
 1011  C-NZ
 1100  CV--
 1101  CV-Z
 1110  CVN-
 1111  CVNZ

Addressing Modes for "a" and "b".

 00  Register Direct
 01  Register Indirect
 10  Register Indirect with Pre-decrement
 11  Register Indirect with Post-increment


Register Numbers for "a" and "b".

 0000  Register 0
 0001  Register 1
 0010  Register 2
 0011  Register 3
 0100  Register 4
 0101  Register 5
 0110  Register 6
 0111  Register 7
 1000  Register 8
 1001  Register 9
 1010  Register 10
 1011  Register 11
 1100  Register 12
 1101  Register 13
 1110  Register 14
 1111  Register 15

|#



;;;  Note:  Different names are used for INTEGERP, INT-TO-V, and
;;;  V-TO-INT (INTP, INT-TO-BV, and BV-TO-INT) so there are no
;;;  name clashes with the conversion functions in the file
;;;  "alu-interpretation.events".  The function ADD produces
;;;  the sum of two integers and SUB subtracts two integers.
;;;  EXP2 multiplies 2 by Y times.


(defn intp (x)
  (or (numberp x) (negativep x)))
```

```
(defn int-to-bv (x size)        ; Integer to two's complement bit vector
  (if (negativep x)
      (v-subtracter-output f
                           (nat-to-v (negative-guts x) size)
                           (nat-to-v 0 size))
      (nat-to-v x size)))


(defn bv-to-int (x)
  (if (nth (sub1 (length x)) x)
      (minus (v-to-nat (v-inc (v-not x))))
      (v-to-nat x)))


(defn c10-tf (x)
  (if (nlistp x)
      nil
    (cons (if (equal (car x) 0) f t)
          (c10-tf (cdr x)))))


(defn c10-int (x)
  (bv-to-int (c10-tf x)))


(defn add (x y)
  (if (negativep x)
      (if (negativep y)
          (minus (plus (negative-guts x) (negative-guts y)))
          (if (lessp y (negative-guts x))
              (minus (difference (negative-guts x) y))
              (difference y (negative-guts x))))
      (if (negativep y)
          (if (lessp x (negative-guts y))
              (minus (difference (negative-guts y) x))
              (difference x (negative-guts y)))
          (plus x y))))



(defn integer-minus (x)
  (if (negativep x)
      (negative-guts x)
      (if (zerop x) 0 (minus x))))


(defn sub (x y)
  (add x (integer-minus y)))
```

```
(defn exp2 (y)
  (if (zerop y)
      1
      (times 2 (exp2 (sub1 y)))))


(prove-lemma quotient-stuff (rewrite)
  (implies (and (not (equal n 0))
                (numberp n))
           (lessp (quotient n 2) n)))


(defn log2 (n)
  (if (or (zerop n)
          (equal n 1))
      0
      (add1 (log2 (quotient n 2)))))



;;;  We assume R15 is PC.  We have defined R13 to be a link register
;;;  and R14 to be a stack pointer much like the MC680x0 family.
;;;  The first group below is the way to write register direct.
```

```
(defn asm-register (x)
  (case x
    (r0    #v000000)
    (r1    #v000001)
    (r2    #v000010)
    (r3    #v000011)
    (r4    #v000100)
    (r5    #v000101)
    (r6    #v000110)
    (r7    #v000111)
    (r8    #v001000)
    (r9    #v001001)
    (r10   #v001010)
    (r11   #v001011)
    (r12   #v001100)

    (r13   #v001101)
    (lnk   #v001101)
    (r14   #v001110)
    (tos   #v001110)
    (r15   #v001111)
    (pc    #v001111)


;;;  The register indirect group is below.

    ((r0)  #v010000)
    ((r1)  #v010001)
    ((r2)  #v010010)
    ((r3)  #v010011)
    ((r4)  #v010100)
    ((r5)  #v010101)
    ((r6)  #v010110)
    ((r7)  #v010111)
    ((r8)  #v011000)
    ((r9)  #v011001)
    ((r10) #v011010)
    ((r11) #v011011)
    ((r12) #v011100)

    ((r13) #v011101)
    ((lnk) #v011101)
    ((r14) #v011110)
    ((tos) #v011110)
    ((r15) #v011111)
    ((pc)  #v011111)


;;;  The register indirect with pre-decrement group is below.

    ((r0-)  #v100000)
    ((r1-)  #v100001)
    ((r2-)  #v100010)
    ((r3-)  #v100011)
```

```
    ((r4-)  #v100100)
    ((r5-)  #v100101)
    ((r6-)  #v100110)
    ((r7-)  #v100111)
    ((r8-)  #v101000)
    ((r9-)  #v101001)
    ((r10-) #v101010)
    ((r11-) #v101011)
    ((r12-) #v101100)

    ((r13-) #v101101)
    ((lnk-) #v101101)
    ((r14-) #v101110)
    ((tos-) #v101110)
    ((r15-) #v101111)
    ((pc-)  #v101111)


;;;  The register indirect with post-increment group is below.

    ((r0+)  #v110000)
    ((r1+)  #v110001)
    ((r2+)  #v110010)
    ((r3+)  #v110011)
    ((r4+)  #v110100)
    ((r5+)  #v110101)
    ((r6+)  #v110110)
    ((r7+)  #v110111)
    ((r8+)  #v111000)
    ((r9+)  #v111001)
    ((r10+) #v111010)
    ((r11+) #v111011)
    ((r12+) #v111100)

    ((r13+) #v111101)
    ((lnk+) #v111101)
    ((r14+) #v111110)
    ((tos+) #v111110)
    ((r15+) #v111111)
    ((pc+)  #v111111)

    (otherwise #v000000)))


;;;  A register is either one of the above or a nine-bit immediate
;;;  value.
```

```
(defn asm-register-a (x)
  (if (intp x)
      (append (int-to-bv x 9) (list t))
      (append (asm-register x)
              #v0000)))



;;;  There are 16 operation codes.


(defn asm-op-code (x)
  (case x
    (move #v0000)
    (inc  #v0001)
    (addc #v0010)
    (add  #v0011)
    (neg  #v0100)
    (dec  #v0101)
    (subb #v0110)
    (sub  #v0111)
    (ror  #v1000)
    (asr  #v1001)
    (lsr  #v1010)
    (xor  #v1011)
    (or   #v1100)
    (and  #v1101)
    (not  #v1110)
    (m15  #v1111)
    (otherwise #v0000)))

;;;  The result of each instruction is based on the settings of the
;;;  flags and the conditional storage field.
```

```
(defn asm-store-cc (x)
  (case x
    (cc #v0000)
    (cs #v0001)
    (vc #v0010)
    (vs #v0011)
    (pl #v0100) (nc #v0100)
    (mi #v0101) (ns #v0101)
    (ne #v0110) (zc #v0110)
    (eq #v0111) (zs #v0111)
    (hi #v1000)
    (ls #v1001)
    (ge #v1010)
    (lt #v1011)
    (gt #v1100)
    (le #v1101)
    (t  #v1110)
    (f  #v1111)
    (otherwise #v0000)))

;;;  The contents of the flag registers may be individually updated
;;;  each instruction.


(defn asm-flags (x)
  (let ((flags (unpack x))
        (c     (car (unpack 'c)))
        (v     (car (unpack 'v)))
        (n     (car (unpack 'n)))
        (z     (car (unpack 'z))))
    (if (equal x 't)
        #v1111
      (list (member z flags)
            (member n flags)
            (member v flags)
            (member c flags)))))



;;;  This assember is works in two passes.  The first pass determines
;;;  the values for the labels and the second actually assembles the
;;;  code.  The function ASM is the assembler.   ASM-LINE assembles
;;;  a single line of assembler code.
```

```
(defn resolve-names (list cnt alist)
  (if (nlistp list)
      alist
      (if (litatom (car list))
          (resolve-names (cdr list) cnt (cons (cons (car list) cnt) alist))
          (resolve-names (cdr list) (add1 cnt) alist))))

;;;  The function update-list, when used with a large program, requires
;;;  that the Nqthm variable REDUCE-TERM-CLOCK be set to some "large
;;;  enough" value.


(defn update-list (list alist)
  (if (nlistp list)
      nil
      (cons (if (and (listp (car list))
                     (equal (caar list) 'value))
                (eval$ t (cadar list) alist)
                (car list))
            (update-list (cdr list) alist))))


(defn asm-line (line)
  (let ((op-code  (car line))
        (store-cc (cadr line))
        (flags    (caddr line))
        (regb     (cadddr line))
        (rega     (caddddr line)))
    (if (intp line)
        (int-to-bv line 32)
        (append (asm-register-a rega)
                (append (asm-register regb)
                        (append (asm-flags flags)
                                (append (asm-store-cc store-cc)
                                        (append (asm-op-code op-code)
                                                #v0000))))))))


(defn asm-list (list)
  (if (nlistp list)
      nil
      (if (litatom (car list))
          (asm-list (cdr list))
          (cons (asm-line (car list))
                (asm-list (cdr list))))))
```

```
(defn asm (list)
  (asm-list (update-list list (resolve-names list 0 nil))))



;;;  Below are some utility functions for FM9001 memory and register
;;;  state input and printing.


(defn t-and-f-to-1-and-0 (list)
  (if (nlistp list)
      nil
    (cons (if (falsep (car list)) 0 1)
          (t-and-f-to-1-and-0 (cdr list)))))


(defn asm-to-1-and-0 (list)
  (if (nlistp list)
      nil
    (cons (t-and-f-to-1-and-0  (car list))
          (asm-to-1-and-0 (cdr list)))))


(defn v-to-nat-all (x)
  (if (nlistp x)
      nil
      (cons (v-to-nat (car x))
            (v-to-nat-all (cdr x)))))


(defn v-to-0s-through-Fs (x collect)
  (if (nlistp x)
      collect
      (v-to-0s-through-Fs (cddddr x)
                          (cons (case  (v-to-nat (firstn 4 x))
                                  (0  (cadr (unpack 'X0)))
                                  (1  (cadr (unpack 'X1)))
                                  (2  (cadr (unpack 'X2)))
                                  (3  (cadr (unpack 'X3)))
                                  (4  (cadr (unpack 'X4)))
                                  (5  (cadr (unpack 'X5)))
                                  (6  (cadr (unpack 'X6)))
                                  (7  (cadr (unpack 'X7)))
                                  (8  (cadr (unpack 'X8)))
                                  (9  (cadr (unpack 'X9)))
                                  (10 (cadr (unpack 'XA)))
                                  (11 (cadr (unpack 'XB)))
                                  (12 (cadr (unpack 'XC)))
                                  (13 (cadr (unpack 'XD)))
                                  (14 (cadr (unpack 'XE)))
                                  (otherwise (cadr (unpack 'XF))))
                                collect))))
```

```
(defn v-to-hex (x)
  (pack (cons
         (car (unpack 'x))
         (v-to-0s-through-Fs x 0))))


(defn v-to-hex-all (x)
  (if (nlistp x)
      nil
      (cons (v-to-hex (car x))
            (v-to-hex-all (cdr x)))))


(defn make-pairs (list)
  (if (or (nlistp list)
          (nlistp (cdr list)))
      nil
      (cons (cons (car list)
                  (cadr list))
            (make-pairs (cddr list)))))


(defn add-ram-marker (list-mem)
  (if (nlistp list-mem)
      nil
    (cons (ram (car list-mem))
          (add-ram-marker (cdr list-mem)))))


(defn list-to-mem3 (mem depth)
  (if (zerop depth)
      mem
      (list-to-mem3 (make-pairs mem) (sub1 depth))))


(defn list-to-mem2 (mem depth)
  (car (list-to-mem3 mem depth)))


(defn list-to-mem1 (mem default)
  (let ((log-size (add1 (log2 (length mem)))))
    (list-to-mem2
     (add-ram-marker
      (append mem
              (make-list (difference (exp2 log-size)
                                     (length mem))
                         default)))
     log-size)))
```

```
(defn stub-right (mem levels default)
  (if (zerop levels)
      mem
      (cons (stub-right mem (sub1 levels) default)
            (stub default))))

;;;  The assembler produces its results as a list.  The acutal
;;;  FM9001 behavioral specification requires a tree-based memory
;;;  implementation.  The function LIST-TO-MEM converts a memory
;;;  list to a memory tree.


(defn list-to-mem (mem size default)
  (if (lessp size (add1 (log2 (length mem))))
      (list 'list-to-mem= 'insufficient-size)
      (stub-right (list-to-mem1 mem default)
                  (difference size (add1 (log2 (length mem))))
                  default)))


(defn list-to-tree-mem (mem)
  (list-to-mem mem 32 (make-list 32 f)))


(defn mem-to-list (mem)
  (cond ((listp mem) (append (mem-to-list (car mem))
                             (mem-to-list (cdr mem))))
        ((stubp mem) nil)
        ((ramp mem) (list (ram-guts mem)))
        ((romp mem) (list (rom-guts mem)))
        (t mem)))



;;;  UN-FM9001 -- Print the state of the FM9001
```

```
(defn un-fm9001 (state)
  (let ((p-state (car state))
        (mem     (cadr state)))
    (let ((regs   (regs p-state))
          (flags  (flags p-state)))
      (let ((z-flag (z-flag flags))
            (n-flag (n-flag flags))
            (v-flag (v-flag flags))
            (c-flag (c-flag flags)))
        (list
         (list (v-to-nat-all (mem-to-list regs))
               z-flag
               n-flag
               v-flag
               c-flag)
         (v-to-hex-all (mem-to-list mem)))))))


;;;  EXECUTE-FM9001 -- Run a program, loaded at 0, for n steps, with
;;;  an initial set of cleared registers, and cleared flags.


(defn execute-fm9001 (n tree-pgm)
  (un-fm9001
   (fm9001
    (list (list (list-to-mem (make-list 15 (nat-to-v 0 32))
                             4 (make-list 32 f))
                ;;    Z N V C
                (list t f f f))
          tree-pgm)
    n)))
```

```
(defn asm-and-fm9001 (n program)
  (execute-fm9001 n (list-to-mem (asm program)
                                 32 (make-list 32 f))))



#|

;;;  Some sample FM9001 assembler code programs.  Each instruction is
;;;  written as a list with five elements: the op-code, the
;;;  conditional store op-code, the flag registers to update, register
;;;  B, and register A.

;;;  Add 1 + 1

;;;                  st update
;;;           opcode cc cvnz regB   regA

;;;                  s
;;;             o    t
;;;             p    o  u
;;;             -    r  p f  r      r
;;;             c    e  d l  e      e
;;;             o    -  a a  g      g
;;;             d    c  t g  -      -
;;;             e    c  e s  b      a

(setq program
          '((move t  t    r0     (pc+))
             1
            (move t  t    r1     (pc+))
             1
            (add  t  t    r1     r0)))


;;  Load -1 and increment twice


(setq program
          '((move t  t    r0     (pc+))
             -1
            (inc  t  t    r0     r0)
            (inc  t  t    r0     r0)))


;;  Add to memory.
```

```
(setq program
      '((move t t r0 (pc+))            ;0
        4                              ;1
        (add t t (r0) (pc+))           ;2
        3                              ;3
        4))                            ;4 -- Should have 7 at end.

;;;  For the programs just above...

;;;  RESOLVE-NAMES is used to make an association list of labels and
;;;  their values.  UPDATE replaces the labels in an assembler code
;;;  program with their values.


(setq alist (resolve-names program 0 nil))

(setq update (update-list program alist))

;;;  ASM assembles a program and produces a linear memory.  A
;;;  tree-based memory suitable for the FM9001 can be make with
;;;  the function TREE-MEM.


(setq memory (asm program))

(setq tree-mem (list-to-mem memory 32 (make-list 32 f)))

;;;  EXECUTE-FM9001 executes a specified number of instructions.


(execute-fm9001 3 (list-to-mem memory 32 (make-list 32 f)))
```

```
(execute-fm9001 3 tree-mem)



;;;  We now give an example of some code similar to that generated by a
;;;  680x0 assembler.

;;;  Register usage is as follows.  (This usage is
;;;  similar to the allocation of registers on the
;;;  68000 for compiled C code.)

;;;  R15  (PC)   --  program counter
;;;  R14  (TOS)  --  stack pointer
;;;  R13  (LNK)  --  subroutine link register

;;;  R12         --  temporary register, must be restored if used
;;;  ...
;;;  R4          --  temporary register, must be restored if used

;;;  R3          --  subroutine argument 3, restore not necessary
;;;  R2          --  subroutine argument 2, restore not necessary
;;;  R1          --  subroutine argument 1, restore not necessary
;;;  R0          --  subroutine argument 0, return result

;;;  Test Programs for the FM9001

;;;  Multiply two natural numbers: R0 and R1

;;;                  st update
;;;            opcode cc cvnz regB   regA

;;;                  s
;;;              o   t
;;;              p   o  u
;;;              -   r  p f  r      r
;;;              c   e  d l  e      e
;;;              o   -  a a  g      g
;;;              d   c  t g  -      -
;;;              e   c  e s  b      a
```

```
(setq test
      '(test     (move t  f    tos    (pc+))    ; Initialize SP
                 (value stack)
                 (move t  f    lnk    tos)      ; Initialize Link Ptr

                 (move t  f    r0     11)
                 (move t  f    r1     99)

                 (move t  f    (tos-) (pc+))    ; Call MULTIPLY
                 (value test-back)
                 (move t  f    pc     (pc))
                 (value multiply)

       test-back
       end       (move t  f    pc     (pc))
                 (value end)))



(setq multiply
      '(multiply (xor  t  f    r2     r2)       ; Clear R2
                 (move t  f    r3     (pc+))    ; Loop count
                 8

        mul-loop (lsr  t  c    r1     r1)       ; Right shift R1
                 (add  cs f    r2     r0)       ; Add if CS
                 (add  t  f    r0     r0)       ; Left shift R0

                 (lsr  t  c    r1     r1)       ; Right shift R1
                 (add  cs f    r2     r0)       ; Add if CS
                 (add  t  f    r0     r0)       ; Left shift R0

                 (lsr  t  c    r1     r1)       ; Right shift R1
                 (add  cs f    r2     r0)       ; Add if CS
                 (add  t  f    r0     r0)       ; Left shift R0

                 (lsr  t  cz   r1     r1)       ; Right shift R1
                 (add  cs f    r2     r0)       ; Add if CS
                 (add  t  f    r0     r0)       ; Left shift R0

                 (move zs f    pc     (pc+))
                 (value mul-end)

                 (dec  t  z    r3     r3)       ; Decrement loop count
                 (move zc f    pc     (pc+))
                 (value mul-loop)

        mul-end  (move t  f    r0     r2)
                 (move t  f    pc     (tos+))
                 ))
```

```
(setq stack
      '(top-stack 0 0 0 0 0 0 0 0 0 0
                  0 0 0 0 0 0 0 0 0 0
                  0 0 0 0 0 0 0 0 0 0
                  0 0 0 0 0 0 0 0 0 0
                  0 0 0 0 0 0 0 0 0 0
                  0 0 0 0 0 0 0 0 0 0
                  0 0 0 0 0 0 0 0 0 0
                  0 0 0 0 0 0 0 0 0 0
                  0 0 0 0 0 0 0 0 0 0
                  0 0 0 0
        stack))


(setq program (append test (append multiply stack)))


(setq alist (resolve-names program 0 nil))

(setq update (update-list program alist))


(setq memory (asm program))

(setq tree-mem (list-to-mem memory 32 (make-list 32 f)))

(execute-fm9001 4 tree-mem)



;;;  Factorial

;;;                    st update
;;;             opcode cc cvnz regB   regA

;;;                    s
;;;               o    t
;;;               p    o  u
;;;               -    r  p f  r      r
;;;               c    e  d l  e      e
;;;               o    -  a a  g      g
;;;               d    c  t g  -      -
;;;               e    c  e s  b      a
```

```
(setq test
      '(test     (move t  f    tos    (pc+))    ; Initialize SP
                 (value stack)
                 (move t  f    lnk    tos)      ; Initialize Link Ptr

                 (move t  f    (tos-) 12)

                 (move t  f    (tos-) (pc+))    ; Call FACT
                 (value test-back)
                 (move t  f    pc     (pc))
                 (value fact)

       test-back
       end       (move t  f    pc     (pc))
                 (value end)))


(setq fact
      '(fact     (move t  f    (tos-) lnk)      ; Link r14,0
                 (move t  f    lnk    tos)
                 (add  t  f    tos    0)

                 (move t  f    r2     2)        ; X-1  -> R0
                 (add  t  f    r2     lnk)
                 (dec  t  cvzn r0     (r2))

                 (move le f    pc     (pc+))    ; X-1 <= 0, jump
                 (value fact-1)

                 (move t  f    (tos-) r0)       ; Push X-1

                 (move t  f    (tos-) (pc+))    ; Push return address
                 (value fact-back)
                 (move t  f    pc     (pc))     ; Call fact
                 (value fact)

       fact-back (move t  f    r2     2)        ; X  -> R1
                 (add  t  f    r2     lnk)
                 (move t  f    r1     (r2))

                 (move t  f    (tos-) (pc+))
                 (value fact-end)
                 (move t  f    pc     (pc))
                 (value multiply)

        fact-1   (move t  f    r0     1)

        fact-end (move t  f    tos    lnk)
                 (move t  f    lnk    (tos+))
                 (move t  f    pc     (tos+))))
```

```
(setq program (append test
                      (append fact
                              (append multiply stack))))


(setq alist (resolve-names program 0 nil))

(setq update (update-list program alist))


(setq memory (asm program))

(setq tree-mem (list-to-mem memory 32 (make-list 32 f)))

(execute-fm9001 4 tree-mem)


;;;  Less than top-of-stack.


(setq less-than
      '(less-than (sub t  cvnz (tos+)  (tos))
                  (xor t  f    (tos)   (tos))
                  (inc le f    (tos)   (tos))))




(r-loop)


|#
```

14.56 "store-resultp.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    STORE-RESULTP.EVENTS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  An inverting 7-to-1 mux, with an implied F for the 8th data input.


(defn store-resultp-mux (s0 s1 s2 d0 d1 d2 d3 d4 d5 d6)
  (let ((s0- (b-not s0))
        (s1- (b-not s1))
        (s2- (b-not s2)))
    (let ((x01 (ao2 s0- d0 s0 d1))
          (x23 (ao2 s0- d2 s0 d3))
          (x45 (ao2 s0- d4 s0 d5))
          (x67 (b-nand s0- d6)))
      (let ((x0123 (ao2 s1- x01 s1 x23))
            (x4567 (ao2 s1- x45 s1 x67)))
        (ao2 s2- x0123 s2 x4567)))))


(defn-to-module store-resultp-mux)

;;; Alternate definition of store-resultp.


(defn b-store-resultp (store-cc flags)
  (let ((s0 (car store-cc))
        (s1 (cadr store-cc))
        (s2 (caddr store-cc))
        (s3 (cadddr store-cc))
        (z (car flags))
        (n (cadr flags))
        (v (caddr flags))
        (c (cadddr flags)))
    (b-xor s0 (store-resultp-mux
               s1 s2 s3
               c v n z
               (b-or c z) (b-xor n v) (b-or z (b-xor n v))))))


(disable b-store-resultp)
```

```
(defn f$b-store-resultp (store-cc flags)
  (let ((s0 (car store-cc))
        (s1 (cadr store-cc))
        (s2 (caddr store-cc))
        (s3 (cadddr store-cc))
        (z (car flags))
        (n (cadr flags))
        (v (caddr flags))
        (c (cadddr flags)))
    (f-xor s0 (f$store-resultp-mux
               s1 s2 s3
               c v n z
               (f-or c z) (f-xor n v) (f-or z (f-xor n v))))))


(disable f$b-store-resultp)


(prove-lemma f$b-store-resultp=b-store-resultp (rewrite)
  (implies
   (and
    (bvp store-cc) (equal (length store-cc) 4)
    (bvp flags) (equal (length flags) 4))
  (equal (f$b-store-resultp store-cc flags)
         (b-store-resultp store-cc flags)))
  ;;Hint
  ((enable f$b-store-resultp b-store-resultp boolp-b-gates bvp-length)
   (disable-theory f-gates)))


(defn b-store-resultp* ()
  '(b-store-resultp (s0 s1 s2 s3 z n v c) (result)
                    ((g0 (cz) b-or (c z))
                     (g1 (nv) b-xor (n v))
                     (g2 (znv) b-or (z nv))
                     (g3 (mux) store-resultp-mux (s1 s2 s3 c v n z cz nv znv))
                     (g4 (result) b-xor (s0 mux)))
                    nil))


(module-predicate b-store-resultp*)


(module-netlist b-store-resultp*)

;;;  Proof of equivalence.
```

```
(prove-lemma b-store-resultp=store-resultp$help (rewrite)
  (implies
   (and
    (boolp s0) (boolp s1) (boolp s2) (boolp s3)
    (boolp s4) (boolp s5) (boolp s6) (boolp s7))
   (equal (b-store-resultp (list s0 s1 s2 s3) (list s4 s5 s6 s7))
          (store-resultp (list s0 s1 s2 s3) (list s4 s5 s6 s7))))
  ;;Hint
  ((enable b-store-resultp store-resultp boolp)))


(prove-lemma b-store-resultp=store-resultp (rewrite)
  (implies
   (and
    (bvp store-cc) (equal (length store-cc) 4)
    (bvp flags) (equal (length flags) 4))
   (equal (b-store-resultp store-cc flags)
          (store-resultp store-cc flags)))
  ;;Hint
  ((enable bvp equal-length-add1)))


(prove-lemma b-store-resultp$value (rewrite)
  (implies
   (and (b-store-resultp& netlist)
        (properp store-cc) (equal (length store-cc) 4)
        (properp store-cc) (equal (length flags) 4))
   (equal (dual-eval 0 'b-store-resultp
                     (append store-cc flags)
                     state netlist)
          (list (f$b-store-resultp store-cc flags))))
  ;;Hint
  ((enable b-store-resultp& f$b-store-resultp
           store-resultp-mux$value b-or$value b-xor$value
           equal-length-add1)
   (disable-theory f-gates)))
```

14.57 "control-modules.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    CONTROL-MODULES.EVENTS
;;;
;;;    Small modules used to define the control logic.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;     UNARY-OP-CODE-P
;;;
;;;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn f$unary-op-code-p (op)
  (let ((op0 (car op))
        (op1 (cadr op))
        (op2 (caddr op))
        (op3 (cadddr op)))
    (let ((op0- (f-not op0))
          (op1- (f-not op1))
          (op2- (f-not op2))
          (op3- (f-not op3)))
      (let ((s0 (f-nand op3- op1-))
            (s1 (f-nand op2- op1-))
            (s2 (f-nand3 op3 op1 op0-))
            (s3 (f-nand3 op3 op2 op1)))
        (f-nand4 s0 s1 s2 s3)))))


(disable f$unary-op-code-p)


(prove-lemma f$unary-op-code-p=unary-op-code-p (rewrite)
  (implies
   (and
    (bvp op) (equal (length op) 4))
   (equal (f$unary-op-code-p op)
          (unary-op-code-p op)))
  ;;Hint
  ((enable f$unary-op-code-p unary-op-code-p bvp-length bvp equal-length-add1)
   (disable-theory f-gates)))
```

```
(defn unary-op-code-p* ()
  '(unary-op-code-p (op0 op1 op2 op3) (z)
                    ((g0 (op0-) b-not   (op0))
                     (g1 (op1-) b-not   (op1))
                     (g2 (op2-) b-not   (op2))
                     (g3 (op3-) b-not   (op3))
                     (g4 (s0)   b-nand  (op3- op1-))
                     (g5 (s1)   b-nand  (op2- op1-))
                     (g6 (s2)   b-nand3 (op3 op1 op0-))
                     (g7 (s3)   b-nand3 (op3 op2 op1))
                     (g8 (z)    b-nand4 (s0 s1 s2 s3)))
                    nil))


(module-predicate unary-op-code-p*)


(module-netlist unary-op-code-p*)


(prove-lemma unary-op-code-p$value (rewrite)
  (implies
   (unary-op-code-p& netlist)
   (equal (dual-eval 0 'unary-op-code-p op-code state netlist)
          (list (f$unary-op-code-p op-code))))
  ;;Hint
  ((enable unary-op-code-p& f$unary-op-code-p
           b-not$value
           b-nand$value b-nand3$value b-nand4$value list-elim-4)
   (disable-theory f-gates)))


(disable unary-op-code-p$value)

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    DECODE-REG-MODE
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn f$decode-reg-mode (m)
  (let ((m0 (car m))
        (m1 (cadr m)))
    (list (f-nor m0 m1)
          (f-nor m0 (f-not m1))
          (id m1))))


(disable f$decode-reg-mode)
```

```
(defn decode-reg-mode* ()
  '(decode-reg-mode (m0 m1) (direct pre-dec side-effect)
                    ((g0 (direct) b-nor (m0 m1))
                     (g1 (m1-) b-not (m1))
                     (g2 (pre-dec) b-nor (m0 m1-))
                     (g3 (side-effect) id (m1)))
                    nil))


(module-predicate decode-reg-mode*)


(module-netlist decode-reg-mode*)


(prove-lemma decode-reg-mode$value (rewrite)
  (implies
   (decode-reg-mode& netlist)
   (equal (dual-eval 0 'decode-reg-mode mode state netlist)
          (f$decode-reg-mode mode)))
  ;;Hint
  ((enable decode-reg-mode& b-nor$value id$value b-not$value
           f$decode-reg-mode)))


(disable decode-reg-mode$value)


(prove-lemma f$decode-reg-mode-as-reg-mode (rewrite)
  (implies
   (and (bvp mode)
        (equal (length mode) 2))
   (equal (f$decode-reg-mode mode)
          (list (reg-direct-p mode)
                (pre-dec-p mode)
                (or (pre-dec-p mode)
                    (post-inc-p mode)))))
  ;;Hint
  ((enable-theory reg-mode-theory)
   (enable f$decode-reg-mode equal-length-add1 bvp)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    SELECT-OP-CODE
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defn f$select-op-code (select dec op)
  (let ((op0 (car op))
        (op1 (cadr op))
        (op2 (caddr op))
        (op3 (cadddr op)))
    (list (f-nand select (f-not op0))
          (f-and select op1)
          (f-if select op2 dec)
          (f-and select op3))))


(disable f$select-op-code)


(prove-lemma properp-length-f$select-op-code (rewrite)
  (and (properp (f$select-op-code select dec op))
       (equal (length (f$select-op-code select dec op)) 4))
  ;;Hint
  ((enable f$select-op-code)
   (disable-theory f-gates)))


(prove-lemma f$select-op-code-selects (rewrite)
  (implies
   (and (boolp select)
        (boolp dec)
        (bvp op)
        (equal (length op) 4))
   (equal (f$select-op-code select dec op)
          (v-if select
                op
                (v-if dec
                      (alu-dec-op)
                      (alu-inc-op)))))
  ;;Hint
  ((enable f$select-op-code bvp-length equal-length-add1 boolp-b-gates
           alu-inc-op alu-dec-op v-if)
   (disable-theory f-gates)))


(defn select-op-code* ()
  '(select-op-code (select dec op0 op1 op2 op3) (z0 z1 z2 z3)
                   ((i0 (op0-) b-not (op0))
                    (g0 (z0)   b-nand (select op0-))
                    (g1 (z1)   b-and  (select op1))
                    (g2 (z2)   b-if   (select op2 dec))
                    (g3 (z3)   b-and  (select op3)))
                   nil))


(module-predicate select-op-code*)
```

```
(module-netlist select-op-code*)


(prove-lemma select-op-code$value (rewrite)
  (implies
   (and (select-op-code& netlist)
        (properp op)
        (equal (length op) 4))
   (equal (dual-eval 0 'select-op-code (list* select dec op) state netlist)
          (f$select-op-code select dec op)))
  ;;Hint
  ((enable select-op-code& f$select-op-code bvp-length equal-length-add1
           b-not$value b-and$value b-nand$value b-if$value)
   (disable-theory f-gates)))


(disable select-op-code$value)

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    V-IF-F-4
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn v-if-f-4* ()
  '(v-if-f-4 (c a0 a1 a2 a3) (z0 z1 z2 z3)
             ((cb (c-) b-not (c))
              (g0 (z0) b-and (c- a0))
              (g1 (z1) b-and (c- a1))
              (g2 (z2) b-and (c- a2))
              (g3 (z3) b-and (c- a3)))
             nil))


(module-predicate v-if-f-4*)


(module-netlist v-if-f-4*)


(defn f$v-if-f-4 (c a)
  (list (f-and (f-not c) (car a))
        (f-and (f-not c) (cadr a))
        (f-and (f-not c) (caddr a))
        (f-and (f-not c) (cadddr a))))


(disable f$v-if-f-4)
```

```
(prove-lemma properp-length-f$v-if-f-4 (rewrite)
  (and (properp (f$v-if-f-4 c a))
       (equal (length (f$v-if-f-4 c a)) 4))
  ;;Hint
  ((enable f$v-if-f-4)
   (disable-theory f-gates)))


(prove-lemma v-if-f-4$value (rewrite)
  (implies
   (v-if-f-4& netlist)
   (equal (dual-eval 0 'v-if-f-4 (cons c a) state netlist)
          (f$v-if-f-4 c a)))
  ;;Hint
  ((enable v-if-f-4& b-not$value b-and$value f$v-if-f-4)
   (disable-theory f-gates)))


(prove-lemma f$v-if-f-4=fv-if (rewrite)
  (implies
   (boolp c)
   (equal (f$v-if-f-4 c a)
          (fv-if c (make-list 4 f) a)))
  ;;Hint
  ((enable f$v-if-f-4 fv-if boolp-b-gates)))


(prove-lemma v-if-f-4$reset-value (rewrite)
  (implies
   (v-if-f-4& netlist)
   (equal (dual-eval 0 'v-if-f-4 (list* t a) state netlist)
          (make-list 4 f)))
  ;;Hint
  ((enable v-if-f-4& b-not$value b-and$value)))


(disable v-if-f-4$reset-value)

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    FANOUT-4
;;;    FANOUT-5
;;;    FANOUT-32
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defn fanout-4* ()
  '(fanout-4 (a) (z0 z1 z2 z3)
             ((aa (aa) b-buf (a))
              (g0 (z0) id (aa))
              (g1 (z1) id (aa))
              (g2 (z2) id (aa))
              (g3 (z3) id (aa)))
             nil))

(module-predicate fanout-4*)

(module-netlist fanout-4*)

(prove-lemma fanout-4$value (rewrite)
  (implies
   (fanout-4& netlist)
   (equal (dual-eval 0 'fanout-4 (list a) state netlist)
          (make-list 4 (threefix a))))
  ;;Hint
  ((enable fanout-4& b-buf$value id$value open-make-list)))

(disable fanout-4$value)

(defn fanout-5* ()
  '(fanout-5 (a) (z0 z1 z2 z3 z4)
             ((aa (aa) b-buf (a))
              (g0 (z0) id (aa))
              (g1 (z1) id (aa))
              (g2 (z2) id (aa))
              (g3 (z3) id (aa))
              (g4 (z4) id (aa)))
             nil))

(module-predicate fanout-5*)

(module-netlist fanout-5*)

(prove-lemma fanout-5$value (rewrite)
  (implies
   (fanout-5& netlist)
   (equal (dual-eval 0 'fanout-5 (list a) state netlist)
          (make-list 5 (threefix a))))
  ;;Hint
  ((enable fanout-5& b-buf$value id$value open-make-list)))
```

```
(disable fanout-5$value)


(defn fanout-32* ()
  '(fanout-32 (a)
              (s0 s1 s2 s3 s4 s5 s6 s7
                  s8 s9 s10 s11 s12 s13 s14 s15
                  s16 s17 s18 s19 s20 s21 s22 s23
                  s24 s25 s26 s27 s28 s29 s30 s31 )
              ((ga (aa) b-buf-pwr (a))
               (g0 (s0) id (aa))
               (g1 (s1) id (aa))
               (g2 (s2) id (aa))
               (g3 (s3) id (aa))
               (g4 (s4) id (aa))
               (g5 (s5) id (aa))
               (g6 (s6) id (aa))
               (g7 (s7) id (aa))
               (g8 (s8) id (aa))
               (g9 (s9) id (aa))
               (g10 (s10) id (aa))
               (g11 (s11) id (aa))
               (g12 (s12) id (aa))
               (g13 (s13) id (aa))
               (g14 (s14) id (aa))
               (g15 (s15) id (aa))
               (g16 (s16) id (aa))
               (g17 (s17) id (aa))
               (g18 (s18) id (aa))
               (g19 (s19) id (aa))
               (g20 (s20) id (aa))
               (g21 (s21) id (aa))
               (g22 (s22) id (aa))
               (g23 (s23) id (aa))
               (g24 (s24) id (aa))
               (g25 (s25) id (aa))
               (g26 (s26) id (aa))
               (g27 (s27) id (aa))
               (g28 (s28) id (aa))
               (g29 (s29) id (aa))
               (g30 (s30) id (aa))
               (g31 (s31) id (aa)))
              nil))


(module-predicate fanout-32*)


(module-netlist fanout-32*)
```

```
(prove-lemma fanout-32$value (rewrite)
  (implies
   (fanout-32& netlist)
   (equal (dual-eval 0 'fanout-32 (list a) state netlist)
          (make-list 32 (threefix a))))
  ;;Hint
  ((enable fanout-32& b-buf-pwr$value id$value open-make-list)))


(disable fanout-32$value)

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    DECODE-5
;;;
;;;    A 5-to-32 , one-hot decoder.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defn f$decode-5 (s)
  (let ((s0 (car s))
        (s1 (cadr s))
        (s2 (caddr s))
        (s3 (cadddr s))
        (s4 (caddddr s)))
    (let ((s0- (f-not s0))
          (s1- (f-not s1))
          (s2- (f-not s2))
          (s3- (f-not s3))
          (s4- (f-not s4)))
      (let ((s0 (f-not s0-))
            (s1 (f-not s1-))
            (s2 (f-not s2-))
            (s3 (f-not s3-))
            (s4 (f-not s4-)))
        (let ((l0 (f-nand s0- s1-))
              (l1 (f-nand s0 s1-))
              (l2 (f-nand s0- s1))
              (l3 (f-nand s0 s1))
              (h0 (f-nand3 s2- s3- s4-))
              (h1 (f-nand3 s2 s3- s4-))
              (h2 (f-nand3 s2- s3 s4-))
              (h3 (f-nand3 s2 s3 s4-))
              (h4 (f-nand3 s2- s3- s4))
              (h5 (f-nand3 s2 s3- s4))
              (h6 (f-nand3 s2- s3 s4))
              (h7 (f-nand3 s2 s3 s4)))
          (let ((x00 (f-nor l0 h0))
                (x10 (f-nor l1 h0))
                (x20 (f-nor l2 h0))
                (x30 (f-nor l3 h0))
                (x01 (f-nor l0 h1))
                (x11 (f-nor l1 h1))
                (x21 (f-nor l2 h1))
                (x31 (f-nor l3 h1))
                (x02 (f-nor l0 h2))
                (x12 (f-nor l1 h2))
                (x22 (f-nor l2 h2))
                (x32 (f-nor l3 h2))
                (x03 (f-nor l0 h3))
                (x13 (f-nor l1 h3))
                (x23 (f-nor l2 h3))
                (x33 (f-nor l3 h3))
                (x04 (f-nor l0 h4))
                (x14 (f-nor l1 h4))
                (x24 (f-nor l2 h4))
                (x34 (f-nor l3 h4))
                (x05 (f-nor l0 h5))
                (x15 (f-nor l1 h5))
                (x25 (f-nor l2 h5))
                (x35 (f-nor l3 h5))
                (x06 (f-nor l0 h6))
                (x16 (f-nor l1 h6))
```

```
            (x26 (f-nor l2 h6))
            (x36 (f-nor l3 h6))
            (x07 (f-nor l0 h7))
            (x17 (f-nor l1 h7))
            (x27 (f-nor l2 h7))
            (x37 (f-nor l3 h7)))
        (list x00 x10 x20 x30 x01 x11 x21 x31 x02 x12 x22 x32 x03 x13 x23
              x33 x04 x14 x24 x34 x05 x15 x25 x35 x06 x16 x26 x36 x07 x17
              x27 x37)))))))

(disable f$decode-5)
```

```
(defn decode-5 (s)
  (let ((s0 (car s))
        (s1 (cadr s))
        (s2 (caddr s))
        (s3 (cadddr s))
        (s4 (caddddr s)))
    (let ((s0- (b-not s0))
          (s1- (b-not s1))
          (s2- (b-not s2))
          (s3- (b-not s3))
          (s4- (b-not s4)))
      (let ((s0 (b-not s0-))
            (s1 (b-not s1-))
            (s2 (b-not s2-))
            (s3 (b-not s3-))
            (s4 (b-not s4-)))
        (let ((l0 (b-nand s0- s1-))
              (l1 (b-nand s0 s1-))
              (l2 (b-nand s0- s1))
              (l3 (b-nand s0 s1))
              (h0 (b-nand3 s2- s3- s4-))
              (h1 (b-nand3 s2 s3- s4-))
              (h2 (b-nand3 s2- s3 s4-))
              (h3 (b-nand3 s2 s3 s4-))
              (h4 (b-nand3 s2- s3- s4))
              (h5 (b-nand3 s2 s3- s4))
              (h6 (b-nand3 s2- s3 s4))
              (h7 (b-nand3 s2 s3 s4)))
          (let ((x00 (b-nor l0 h0))
                (x10 (b-nor l1 h0))
                (x20 (b-nor l2 h0))
                (x30 (b-nor l3 h0))
                (x01 (b-nor l0 h1))
                (x11 (b-nor l1 h1))
                (x21 (b-nor l2 h1))
                (x31 (b-nor l3 h1))
                (x02 (b-nor l0 h2))
                (x12 (b-nor l1 h2))
                (x22 (b-nor l2 h2))
                (x32 (b-nor l3 h2))
                (x03 (b-nor l0 h3))
                (x13 (b-nor l1 h3))
                (x23 (b-nor l2 h3))
                (x33 (b-nor l3 h3))
                (x04 (b-nor l0 h4))
                (x14 (b-nor l1 h4))
                (x24 (b-nor l2 h4))
                (x34 (b-nor l3 h4))
                (x05 (b-nor l0 h5))
                (x15 (b-nor l1 h5))
                (x25 (b-nor l2 h5))
                (x35 (b-nor l3 h5))
                (x06 (b-nor l0 h6))
                (x16 (b-nor l1 h6))
```

```
                (x26 (b-nor l2 h6))
                (x36 (b-nor l3 h6))
                (x07 (b-nor l0 h7))
                (x17 (b-nor l1 h7))
                (x27 (b-nor l2 h7))
                (x37 (b-nor l3 h7)))
            (list x00 x10 x20 x30 x01 x11 x21 x31 x02 x12 x22 x32 x03 x13 x23
                  x33 x04 x14 x24 x34 x05 x15 x25 x35 x06 x16 x26 x36 x07 x17
                  x27 x37)))))))
```

```
(disable decode-5)
```

```
(prove-lemma bvp-length-decode-5 (rewrite)
  (and (bvp (decode-5 s))
       (equal (length (decode-5 s)) 32))
  ;;Hint
  ((enable decode-5 boolp-b-gates)
   (disable-theory b-gates)))
```

```
(defn decode-5* ()
  '(decode-5 (s0 s1 s2 s3 s4)
             (x00 x10 x20 x30 x01 x11 x21 x31 x02 x12 x22 x32
                  x03 x13 x23 x33 x04 x14 x24 x34 x05 x15 x25 x35
                  x06 x16 x26 x36 x07 x17 x27 x37)
             ((gs0- (s0-) b-not   (s0))
              (gs1- (s1-) b-not   (s1))
              (gs2- (s2-) b-not   (s2))
              (gs3- (s3-) b-not   (s3))
              (gs4- (s4-) b-not   (s4))
              (gs0  (bs0)  b-not   (s0-))
              (gs1  (bs1)  b-not   (s1-))
              (gs2  (bs2)  b-not   (s2-))
              (gs3  (bs3)  b-not   (s3-))
              (gs4  (bs4)  b-not   (s4-))
              (gl0  (l0)  b-nand  (s0- s1-))
              (gl1  (l1)  b-nand  (bs0 s1-))
              (gl2  (l2)  b-nand  (s0- bs1))
              (gl3  (l3)  b-nand  (bs0 bs1))
              (gh0  (h0)  b-nand3 (s2- s3- s4-))
              (gh1  (h1)  b-nand3 (bs2 s3- s4-))
              (gh2  (h2)  b-nand3 (s2- bs3 s4-))
              (gh3  (h3)  b-nand3 (bs2 bs3 s4-))
              (gh4  (h4)  b-nand3 (s2- s3- bs4))
              (gh5  (h5)  b-nand3 (bs2 s3- bs4))
              (gh6  (h6)  b-nand3 (s2- bs3 bs4))
              (gh7  (h7)  b-nand3 (bs2 bs3 bs4))
              (gx00 (x00) b-nor   (l0 h0))
              (gx10 (x10) b-nor   (l1 h0))
              (gx20 (x20) b-nor   (l2 h0))
              (gx30 (x30) b-nor   (l3 h0))
              (gx01 (x01) b-nor   (l0 h1))
              (gx11 (x11) b-nor   (l1 h1))
              (gx21 (x21) b-nor   (l2 h1))
              (gx31 (x31) b-nor   (l3 h1))
              (gx02 (x02) b-nor   (l0 h2))
              (gx12 (x12) b-nor   (l1 h2))
              (gx22 (x22) b-nor   (l2 h2))
              (gx32 (x32) b-nor   (l3 h2))
              (gx03 (x03) b-nor   (l0 h3))
              (gx13 (x13) b-nor   (l1 h3))
              (gx23 (x23) b-nor   (l2 h3))
              (gx33 (x33) b-nor   (l3 h3))
              (gx04 (x04) b-nor   (l0 h4))
              (gx14 (x14) b-nor   (l1 h4))
              (gx24 (x24) b-nor   (l2 h4))
              (gx34 (x34) b-nor   (l3 h4))
              (gx05 (x05) b-nor   (l0 h5))
              (gx15 (x15) b-nor   (l1 h5))
              (gx25 (x25) b-nor   (l2 h5))
              (gx35 (x35) b-nor   (l3 h5))
              (gx06 (x06) b-nor   (l0 h6))
              (gx16 (x16) b-nor   (l1 h6))
              (gx26 (x26) b-nor   (l2 h6))
```

```
             (gx36 (x36) b-nor   (l3 h6))
             (gx07 (x07) b-nor   (l0 h7))
             (gx17 (x17) b-nor   (l1 h7))
             (gx27 (x27) b-nor   (l2 h7))
             (gx37 (x37) b-nor   (l3 h7)))
            nil))


(module-predicate decode-5*)


(module-netlist decode-5*)


(prove-lemma properp-length-f$decode-5 (rewrite)
  (and (properp (f$decode-5 s))
       (equal (length (f$decode-5 s)) 32))
  ;;Hint
  ((enable f$decode-5)
   (disable-theory f-gates)))


(prove-lemma decode-5$value (rewrite)
  (implies
   (and (decode-5& netlist)
        (equal (length s) 5))
   (equal (dual-eval 0 'decode-5 s state netlist)
          (f$decode-5 s)))
  ;;Hint
  ((enable f$decode-5 decode-5& equal-length-add1
           b-not$value b-nand$value b-nand3$value b-nor$value)
   (disable-theory f-gates b-gates)
   (disable cons-equal)))


(prove-lemma f$decode-5=decode-5 (rewrite)
  (implies
   (and (bvp s)
        (equal (length s) 5))
   (equal (f$decode-5 s)
          (decode-5 s)))
  ;;Hint
  ((enable f$decode-5 decode-5 boolp-b-gates equal-length-add1 bvp)
   (disable-theory f-gates b-gates)
   (disable cons-equal)))
```

```
(disable decode-5$value)

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    ENCODE-32
;;;
;;;    A 32-to-5 encoder, assuming a one-hot input vector.
;;;
;;;    This encoder is optimized to eliminate encoding of the unused
;;;    states S26, S27, and S31.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn encode-32 (
                 s0 s1 s2 s3 s4 s5 s6 s7
                 s8 s9 s10 s11 s12 s13 s14 s15
                 s16 s17 s18 s19 s20 s21 s22 s23
                 s24 s25 s26 s27 s28 s29 s30 s31)

  (let ((x0a (b-nor4 s1 s3 s5 s7))
        (x0b (b-nor4 s9 s11 s13 s15))
        (x0c (b-nor4 s17 s19 s21 s23))
        (x0d (b-nor  s25 s29))
        (x1a (b-nor4 s2 s3 s6 s7))
        (x1b (b-nor4 s10 s11 s14 s15))
        (x1c (b-nor4 s18 s19 s22 s23))
        (x1d (b-not  s30))
        (x2a (b-nor4 s4 s5 s6 s7))
        (x2b (b-nor4 s12 s13 s14 s15))
        (x2c (b-nor4 s20 s21 s22 s23))
        (x2d (b-nor3 s28 s29 s30))
        (x3a (b-nor4 s8 s9 s10 s11))
        (x3b (b-nor4 s12 s13 s14 s15))
        (x3c (b-nor  s24 s25))
        (x3d (b-nor3 s28 s29 s30))
        (x4a (b-nor4 s16 s17 s18 s19))
        (x4b (b-nor4 s20 s21 s22 s23))
        (x4c (b-nor  s24 s25))
        (x4d (b-nor3 s28 s29 s30)))
    (let ((x0 (b-nand4 x0a x0b x0c x0d))
          (x1 (b-nand4 x1a x1b x1c x1d))
          (x2 (b-nand4 x2a x2b x2c x2d))
          (x3 (b-nand4 x3a x3b x3c x3d))
          (x4 (b-nand4 x4a x4b x4c x4d)))
      (list x0 x1 x2 x3 x4))))


(disable encode-32)


(defn-to-module encode-32)
```

```
(prove-lemma bvp-length-encode-32 (rewrite)
  (and
   (bvp (encode-32 s0 s1 s2 s3 s4 s5 s6 s7
                   s8 s9 s10 s11 s12 s13 s14 s15
                   s16 s17 s18 s19 s20 s21 s22 s23
                   s24 s25 s26 s27 s28 s29 s30 s31))
   (equal (length (encode-32 s0 s1 s2 s3 s4 s5 s6 s7
                             s8 s9 s10 s11 s12 s13 s14 s15
                             s16 s17 s18 s19 s20 s21 s22 s23
                             s24 s25 s26 s27 s28 s29 s30 s31))
          5))
  ;;Hint
  ((enable encode-32)))


(prove-lemma properp-length-f$encode-32 (rewrite)
  (and
   (properp (f$encode-32 s0 s1 s2 s3 s4 s5 s6 s7
                   s8 s9 s10 s11 s12 s13 s14 s15
                   s16 s17 s18 s19 s20 s21 s22 s23
                   s24 s25 s26 s27 s28 s29 s30 s31))
   (equal (length (f$encode-32 s0 s1 s2 s3 s4 s5 s6 s7
                             s8 s9 s10 s11 s12 s13 s14 s15
                             s16 s17 s18 s19 s20 s21 s22 s23
                             s24 s25 s26 s27 s28 s29 s30 s31))
          5))
  ;;Hint
  ((enable f$encode-32)
   (disable-theory f-gates)))
```

```
(prove-lemma dual-eval-on-collected-nth-32 ()
  (implies
   (and (equal (length args) 32)
        (properp args))
   (equal (dual-eval flag name args state netlist)
          (dual-eval flag name
                     (list (nth 0 args)
                           (nth 1 args)
                           (nth 2 args)
                           (nth 3 args)
                           (nth 4 args)
                           (nth 5 args)
                           (nth 6 args)
                           (nth 7 args)
                           (nth 8 args)
                           (nth 9 args)
                           (nth 10 args)
                           (nth 11 args)
                           (nth 12 args)
                           (nth 13 args)
                           (nth 14 args)
                           (nth 15 args)
                           (nth 16 args)
                           (nth 17 args)
                           (nth 18 args)
                           (nth 19 args)
                           (nth 20 args)
                           (nth 21 args)
                           (nth 22 args)
                           (nth 23 args)
                           (nth 24 args)
                           (nth 25 args)
                           (nth 26 args)
                           (nth 27 args)
                           (nth 28 args)
                           (nth 29 args)
                           (nth 30 args)
                           (nth 31 args))
                     state netlist)))
  ;;hint
  ((use (equal-length-32-as-collected-nth (l args)))))
```

```
(prove-lemma encode-32$value-on-a-vector (rewrite)
  (implies
   (and (encode-32& netlist)
        (properp args)
        (equal (length args) 32))
   (equal (dual-eval 0 'encode-32 args state netlist)
          (f$encode-32 (nth 0 args)
                       (nth 1 args)
                       (nth 2 args)
                       (nth 3 args)
                       (nth 4 args)
                       (nth 5 args)
                       (nth 6 args)
                       (nth 7 args)
                       (nth 8 args)
                       (nth 9 args)
                       (nth 10 args)
                       (nth 11 args)
                       (nth 12 args)
                       (nth 13 args)
                       (nth 14 args)
                       (nth 15 args)
                       (nth 16 args)
                       (nth 17 args)
                       (nth 18 args)
                       (nth 19 args)
                       (nth 20 args)
                       (nth 21 args)
                       (nth 22 args)
                       (nth 23 args)
                       (nth 24 args)
                       (nth 25 args)
                       (nth 26 args)
                       (nth 27 args)
                       (nth 28 args)
                       (nth 29 args)
                       (nth 30 args)
                       (nth 31 args))))
  ;;hint
  ((disable f$encode-32 *1*encode-32)
   (enable encode-32$value)
   (use (dual-eval-on-collected-nth-32
         (flag 0)
         (name 'encode-32)
         (args args)
         (state state)
         (netlist netlist)))))


(disable encode-32$value-on-a-vector)
```

14.58 "control.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    CONTROL.EVENTS
;;;
;;;    Many of the events used to define that control logic are created
;;;    automatically, from macros and functions defined in "control.lisp".
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;   Some preliminary definitions.

;;;  SET-SOME-FLAGS


(defn set-some-flags (set-flags)
  (or (z-set set-flags)
      (n-set set-flags)
      (v-set set-flags)
      (c-set set-flags)))


(disable set-some-flags)


(prove-lemma not-set-some-flags-make-list-4-f (rewrite)
  (not (set-some-flags (make-list 4 f))))


(defn flags-hyps (flags)
  (and (equal (length flags) 4)
       (bvp flags)))


(prove-lemma not-set-some-flags-update-flags (rewrite)
  (implies
   (and (flags-hyps flags)
        (not (set-some-flags set-flags)))
   (equal (update-flags flags set-flags cvzbv)
          flags))
  ;;Hint
  ((enable set-some-flags update-flags open-nth
           c-flag v-flag n-flag z-flag
           equal-length-add1)))
```

```
(prove-lemma boolp-c-flag (rewrite)
  (implies
   (flags-hyps flags)
   (boolp (c-flag flags)))
  ;;Hint
  ((enable c-flag)))


(defn f$set-some-flags (set-flags)
  (f-or4 (car set-flags)
         (cadr set-flags)
         (caddr set-flags)
         (cadddr set-flags)))


(disable f$set-some-flags)


(prove-lemma f$set-some-flags=set-some=flags (rewrite)
  (implies
   (and (bvp set-flags)
        (equal (length set-flags) 4))
   (equal (f$set-some-flags set-flags)
          (set-some-flags set-flags)))
  ;;Hint
  ((enable f$set-some-flags set-some-flags equal-length-add1 open-nth
          c-set v-set n-set z-set boolp-b-gates)
   (disable-theory f-gates)))


(prove-lemma b-or4$value-as-f$set-some-flags (rewrite)
  (implies
   (and (b-or4& netlist)
        (properp set-flags)
        (equal (length set-flags) 4))
   (equal (dual-eval 0 'b-or4 set-flags state netlist)
          (list (f$set-some-flags set-flags))))
  ;;Hint
  ((enable b-or4$value equal-length-add1 f$set-some-flags)))


(disable b-or4$value-as-f$set-some-flags)


(defn f$all-t-regs-address (regs-address)
  (f-and4 (car regs-address)
          (cadr regs-address)
          (caddr regs-address)
          (cadddr regs-address)))


(disable f$all-t-regs-address)
```

```
(prove-lemma f$all-t-regs-address=set-some=flags (rewrite)
  (implies
   (and (bvp regs-address)
        (equal (length regs-address) 4))
   (equal (f$all-t-regs-address regs-address)
          (equal regs-address (make-list 4 t))))
  ;;Hint
  ((enable f$all-t-regs-address equal-length-add1 boolp-b-gates open-make-list)
   (disable-theory f-gates)))


(prove-lemma b-and4$value-as-f$all-t-regs-address (rewrite)
  (implies
   (and (b-and4& netlist)
        (properp regs-address)
        (equal (length regs-address) 4))
   (equal (dual-eval 0 'b-and4 regs-address state netlist)
          (list (f$all-t-regs-address regs-address))))
  ;;Hint
  ((enable b-and4& b-and4$value equal-length-add1 f$all-t-regs-address
           properp)))


(disable b-and4$value-as-f$all-t-regs-address)

;;;    CV-HYPS rw- regs-address i-reg flags pc-reg
;;;
;;;    The hyps that allow the CV_state functions to be opened nicely.


(defn cv-hyps (rw- regs-address i-reg flags pc-reg)
  (and (boolp rw-)
       (equal (length regs-address) 4)
       (bvp regs-address)
       (equal (length i-reg) 32)
       (bvp i-reg)
       (flags-hyps flags)
       (equal (length pc-reg) 4)
       (bvp pc-reg)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    MACRO EXECUTION
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;  The first two macros generate numerous simple events.


(define-control-states)
```

```
(define-control-state-accessors)

;;;  This macro creates events that define the control vector for each state.


(define-control-vector-functions)


(prove-lemma cv_fetch1$rw-doan-matta (rewrite)
  (equal (cv_fetch1 F REGS-ADDRESS I-REG FLAGS PC-REG)
         (cv_fetch1 T REGS-ADDRESS I-REG FLAGS PC-REG))
  ;;Hint
  ((enable cv_fetch1)))


(prove-lemma properp-length-cv_fetch1 (rewrite)
  (and
   (implies
    (cv-hyps rw- regs-address i-reg flags pc-reg)
    (equal (length (cv_fetch1 rw- regs-address i-reg flags pc-reg))
           40))
   (properp (cv_fetch1 rw- regs-address i-reg flags pc-reg)))
  ;;Hint
  ((enable cv_fetch1 v_fetch1)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    CONTROL-LET
;;;
;;;    CONTROL-LET computes a number of intermediate results that are used both
;;;    to determine the next control state, and the values of the control
;;;    lines.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defn f$control-let (i-reg flags regs-address)
  (let ((a-immediate-p (a-immediate-p i-reg))
        (mode-a        (mode-a        i-reg))
        (rn-a          (rn-a          i-reg))
        (mode-b        (mode-b        i-reg))
        (rn-b          (rn-b          i-reg))
        (set-flags     (set-flags     i-reg))
        (store-cc      (store-cc      i-reg))
        (op-code       (op-code       i-reg)))
    (let
      ((a-immediate-p (id a-immediate-p)))
      (let
        ((a-immediate-p- (f-not a-immediate-p))
         (store (f$b-store-resultp store-cc flags))
         (set-some-flags (f$set-some-flags set-flags))
         (decode-reg-a-mode (f$decode-reg-mode mode-a))
         (decode-reg-b-mode (f$decode-reg-mode mode-b)))
        (let
          ((almost-direct-a (car decode-reg-a-mode))
           (pre-dec-a (cadr decode-reg-a-mode))
           (almost-side-effect-a (caddr decode-reg-a-mode))
           (direct-b (car decode-reg-b-mode))
           (pre-dec-b (cadr decode-reg-b-mode))
           (side-effect-b (caddr decode-reg-b-mode))
           (unary (f$unary-op-code-p op-code))
           (c (id (c-flag flags)))
           (all-t-regs-address (f$all-t-regs-address regs-address)))
          (let
            ((side-effect-a (f-and a-immediate-p- almost-side-effect-a))
             (direct-a (f-or a-immediate-p almost-direct-a)))
            (list a-immediate-p store set-some-flags direct-a direct-b unary
                  pre-dec-a pre-dec-b c all-t-regs-address
                  side-effect-a side-effect-b)))))))

(disable f$control-let)
```

```
(defn control-let* ()
  (let ((i-reg #i(i-reg 0 32))
        (flags #i(flags 0 4))
        (regs-address #i(regs-address 0 4)))
    (let ((a-immediate-p (a-immediate-p i-reg))
          (mode-a        (mode-a        i-reg))
          (rn-a          (rn-a          i-reg))
          (mode-b        (mode-b        i-reg))
          (rn-b          (rn-b          i-reg))
          (set-flags     (set-flags     i-reg))
          (store-cc      (store-cc      i-reg))
          (op-code       (op-code       i-reg)))
      (list
       'control-let
       (append i-reg (append flags regs-address))
       '(a-immediate-p store set-some-flags direct-a direct-b unary
                       pre-dec-a pre-dec-b c all-t-regs-address
                       side-effect-a side-effect-b)
       (list
        (list 'g0 '(a-immediate-p) 'id (list a-immediate-p))
        (list 'g1 '(a-immediate-p-) 'b-not '(a-immediate-p))
        (list 'g2 '(store) 'b-store-resultp (append store-cc flags))
        (list 'g3 '(set-some-flags) 'b-or4 set-flags)
        (list 'g4 '(almost-direct-a pre-dec-a almost-side-effect-a)
              'decode-reg-mode mode-a)
        (list 'g5 '(direct-b pre-dec-b side-effect-b) 'decode-reg-mode mode-b)
        (list 'g6 '(unary) 'unary-op-code-p op-code)
        (list 'g7 '(c) 'id (list (c-flag flags)))
        (list 'g8 '(all-t-regs-address) 'b-and4 regs-address)
        (list 'g9 '(side-effect-a) 'b-and
              '(a-immediate-p- almost-side-effect-a))
        (list 'ga '(direct-a) 'b-or '(a-immediate-p almost-direct-a)))
       nil))))

(module-predicate control-let*)

(module-netlist control-let*)
```

```
(prove-lemma control-let$value (rewrite)
  (implies
   (and (control-let& netlist)
        (properp i-reg) (equal (length i-reg) 32)
        (properp flags) (equal (length flags) 4)
        (properp regs-address) (equal (length regs-address) 4))
   (equal (dual-eval 0 'control-let (append i-reg (append flags regs-address))
                     state netlist)
          (f$control-let i-reg flags regs-address)))
  ;;Hint
  (#.`(enable f$control-let control-let& nth-indices c-flag
              b-or4$value-as-f$set-some-flags
              b-and4$value-as-f$all-t-regs-address
              ,@(submodule-value-lemmas 'control-let*))
   (disable open-indices *1*indices indices)
   (enable-theory ir-fields-theory)
   (disable-theory f-gates)))


(disable control-let$value)


(prove-lemma f$control-let=control-let (rewrite)
  (implies
   (and (bvp i-reg) (equal (length i-reg) 32)
        (bvp flags) (equal (length flags) 4)
        (bvp regs-address) (equal (length regs-address) 4))
   (equal (f$control-let i-reg flags regs-address)
          #.(control-let
             '(list a-immediate-p store set-some-flags direct-a direct-b unary
                    pre-dec-a pre-dec-b c all-t-regs-address
                    side-effect-a side-effect-b))))
  ;;Hint
  ((enable f$control-let boolp-b-gates expand-*-connectives nth-indices
          bvp-length c-flag)
   (disable open-indices *1*indices indices)
   (enable-theory ir-fields-theory)
   (disable-theory f-gates)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;  NEXT-STATE
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

;;;  The NEXT-STATE logic is synthesized from the *STATE-TABLE* (defined in
;;;  "control.lisp").


(define-next-state)
```

```
(prove-lemma bvp-length-next-state (rewrite)
  (let ((next-state
         (next-state
          decoded-state store set-some-flags unary direct-a direct-b
          side-effect-a side-effect-b all-t-regs-address
          dtack- hold-)))
  (and (equal (length next-state) 32)
       (implies
        (and (bvp decoded-state)
             (equal (length decoded-state) 32))
        (bvp next-state))))

  ;;Hint
  ((enable bvp next-state)))


(prove-lemma properp-length-f$next-state (rewrite)
  (let ((next-state
         (f$next-state
          decoded-state store set-some-flags unary direct-a direct-b
          side-effect-a side-effect-b all-t-regs-address
          dtack- hold-)))
  (and (equal (length next-state) 32)
       (properp next-state)))

  ;;Hint
  ((enable f$next-state)
   (disable-theory f-gates)))


(prove-lemma next-state$value (rewrite)
  (implies
   (and (next-state& netlist)
        (properp decoded-state)
        (equal (length decoded-state) 32))
   (equal (dual-eval 0 'next-state
                     (append decoded-state
                             (list store set-some-flags
                                   unary direct-a direct-b
                                   side-effect-a side-effect-b
                                   all-t-regs-address
                                   dtack- hold-))
                     state netlist)
          (f$next-state decoded-state store set-some-flags
                        unary direct-a direct-b
                        side-effect-a side-effect-b
                        all-t-regs-address
                        dtack- hold-)))
  ;;hint
  (#.`(enable next-state& f$next-state
              ,@(submodule-value-lemmas 'next-state*))
   (disable open-indices *1*indices indices)
   (disable-theory f-gates)))
```

```
(disable next-state$value)

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;   CV
;;;
;;;   CV computes the 40-bit control vector.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#.
```

```
(let ((statelist (mapcar #'car *control-states*)))
  `(DEFN CV (DECODED-STATE
             RN-A RN-B OP-CODE
             PC-REG REGS-ADDRESS SET-FLAGS
             STORE C A-IMMEDIATE-P RW-
             DIRECT-A SIDE-EFFECT-A SIDE-EFFECT-B
             PRE-DEC-A PRE-DEC-B)
     (LET ,(iterate for state in statelist for i from 0
             collect `(,state (NTH ,i DECODED-STATE)))
       (LET ((STORE- (B-NOT STORE)))
         (LET ((ALU-ZERO  (B-OR4 FETCH0 RESET0 RESET1 RESET2))
               (ALU-SWAP  (B-OR3 READB2 WRITE1 SEFB1))
               (INCDECA   (B-OR READA1 SEFA1)))
           (LET ((ALU-OP (V-IF (B-NOR4 FETCH2 INCDECA ALU-SWAP ALU-ZERO)
                               OP-CODE
                               (V-IF (B-NAND (B-NAND INCDECA PRE-DEC-A)
                                             (B-NAND ALU-SWAP PRE-DEC-B))
                                     (ALU-DEC-OP)
                                     (ALU-INC-OP)))))
             (LIST*
              ;;RW-
              (B-AND (B-NAND (B-NOT RW-) FETCH0)
                     (B-NOR3 WRITE1 WRITE2 WRITE3))
              ;;STROBE-
              (B-NOR8 FETCH2 FETCH3 READA2 READA3
                      READB2 READB3 WRITE2 WRITE3)
              ;;HDACK-
              (B-NOT HOLD0)
              ;;WE-REGS
              (B-NAND5 (B-NAND STORE UPDATE)
                       (B-NAND SIDE-EFFECT-A READA1)
                       (B-NAND3 STORE- SIDE-EFFECT-B READB2)
                       (B-NAND SIDE-EFFECT-B WRITE1)
                       (B-NOR5 FETCH2 SEFA1 SEFB1 RESET0 RESET2))
              ;;WE-A-REG
              (B-NAND (B-NAND DIRECT-A READB1)
                      (B-NOR6 FETCH0 REGA READA0 READA3 SEFA0 RESET1))
              ;;WE-B-REG
              (B-NAND (B-NOR4 REGB UPDATE READA2 READB0)
                      (B-NOR4 READB3 WRITE0 SEFB0 RESET1))
              ;;WE-I-REG
              (B-OR FETCH3 RESET1)
              ;;WE-DATA-OUT
              (B-OR WRITE0 RESET0)
              ;;WE-ADDR-OUT
              (B-NAND (B-NOR3 FETCH0 READA0 READB0)
                      (B-NOR WRITE0 RESET1))
              ;;WE-HOLD-
              (B-OR3 FETCH0 HOLD0 RESET0)
              ;;WE-PC-REG
              (B-OR HOLD0 RESET0)
              ;;DATA-IN-SELECT
              (B-OR3 FETCH3 READA3 READB3)
              ;;DEC-ADDR-OUT
```

```
(B-NAND (B-NAND PRE-DEC-A READA0)
        (B-NAND PRE-DEC-B (B-OR READB0 WRITE0)))
;;SELECT-IMMEDIATE
(B-AND A-IMMEDIATE-P (B-OR REGA READB1))
;;ALU-C
(CARRY-IN-HELP (CONS C (CONS ALU-ZERO ALU-OP)))
;;ALU-ZERO
ALU-ZERO
(APPEND
 ;;STATE
 (ENCODE-32 ,@statelist)
 (APPEND
  ;;WE-FLAGS
  (V-IF (B-NOR UPDATE WRITE0)
        (V-IF RESET0
              (MAKE-LIST 4 T)
              (MAKE-LIST 4 F))
        SET-FLAGS)
  (APPEND
   ;;REGS-ADDRESS
   (V-IF (B-NAND (B-NOR3 REGA READA0 READA1)
                 (B-NOR3 READB1 SEFA0 SEFA1))
         RN-A
         (V-IF (B-NAND (B-NOR5 REGB UPDATE READA2 READB0 READB2)
                       (B-NOR4 WRITE0 WRITE1 SEFB0 SEFB1))
               RN-B
               (V-IF (B-OR RESET0 RESET1)
                     (MAKE-LIST 4 F)
                     (V-IF RESET2
                           (V-INC REGS-ADDRESS)
                           PC-REG))))

   (APPEND
    ;;ALU-OP
    ALU-OP
    ;;ALU-MPG
    (MPG (CONS ALU-ZERO (CONS ALU-SWAP ALU-OP)))))))))))))


(disable cv)

#.
```

```
(let ((statelist (mapcar #'car *control-states*)))
  `(DEFN F$CV (DECODED-STATE
               RN-A RN-B OP-CODE
               PC-REG REGS-ADDRESS SET-FLAGS
               STORE C A-IMMEDIATE-P RW-
               DIRECT-A SIDE-EFFECT-A SIDE-EFFECT-B
               PRE-DEC-A PRE-DEC-B)
     (LET ,(iterate for state in statelist for i from 0
              collect `(,state (NTH ,i DECODED-STATE)))
       (LET ((STORE- (F-NOT STORE)))
         (LET ((ALU-ZERO  (F-OR4 FETCH0 RESET0 RESET1 RESET2))
               (ALU-SWAP  (F-OR3 READB2 WRITE1 SEFB1))
               (INCDECA   (F-OR READA1 SEFA1)))
           (LET ((ALU-OP (F$SELECT-OP-CODE
                          (F-NOR4 FETCH2 INCDECA ALU-SWAP ALU-ZERO)
                          (F-NAND (F-NAND INCDECA PRE-DEC-A)
                                  (F-NAND ALU-SWAP PRE-DEC-B))
                          OP-CODE)))

             (LIST*
              ;;RW-
              (F-AND (F-NAND (F-NOT RW-) FETCH0)
                     (F-NOR3 WRITE1 WRITE2 WRITE3))
              ;;STROBE-
              (F-NOR8 FETCH2 FETCH3 READA2 READA3
                      READB2 READB3 WRITE2 WRITE3)
              ;;HDACK-
              (F-NOT HOLD0)
              ;;WE-REGS
              (F-NAND5 (F-NAND STORE UPDATE)
                       (F-NAND SIDE-EFFECT-A READA1)
                       (F-NAND3 STORE- SIDE-EFFECT-B READB2)
                       (F-NAND SIDE-EFFECT-B WRITE1)
                       (F-NOR5 FETCH2 SEFA1 SEFB1 RESET0 RESET2))
              ;;WE-A-REG
              (F-NAND (F-NAND DIRECT-A READB1)
                      (F-NOR6 FETCH0 REGA READA0 READA3 SEFA0 RESET1))
              ;;WE-B-REG
              (F-NAND (F-NOR4 REGB UPDATE READA2 READB0)
                      (F-NOR4 READB3 WRITE0 SEFB0 RESET1))
              ;;WE-I-REG
              (F-OR FETCH3 RESET1)
              ;;WE-DATA-OUT
              (F-OR WRITE0 RESET0)
              ;;WE-ADDR-OUT
              (F-NAND (F-NOR3 FETCH0 READA0 READB0)
                      (F-NOR WRITE0 RESET1))
              ;;WE-HOLD-
              (F-OR3 FETCH0 HOLD0 RESET0)
              ;;WE-PC-REG
              (F-OR HOLD0 RESET0)
              ;;DATA-IN-SELECT
              (F-OR3 FETCH3 READA3 READB3)
              ;;DEC-ADDR-OUT
```

```
              (F-NAND (F-NAND PRE-DEC-A READA0)
                      (F-NAND PRE-DEC-B (F-OR READB0 WRITE0)))
              ;;SELECT-IMMEDIATE
              (F-AND A-IMMEDIATE-P (F-OR REGA READB1))
              ;;ALU-C
              (F$CARRY-IN-HELP (CONS C (CONS ALU-ZERO ALU-OP)))
              ;;ALU-ZERO
              ALU-ZERO
              (APPEND
               ;;STATE
               (F$ENCODE-32 ,@statelist)
               (APPEND
                ;;WE-FLAGS
                (FV-IF (F-NOR UPDATE WRITE0)
                       (MAKE-LIST 4 (THREEFIX RESET0))
                       SET-FLAGS)
                (APPEND
                 ;;REGS-ADDRESS
                 (FV-IF (F-NAND (F-NOR3 REGA READA0 READA1)
                                (F-NOR3 READB1 SEFA0 SEFA1))
                        RN-A
                        (FV-IF
                         (F-NAND (F-NOR5 REGB UPDATE READA2 READB0 READB2)
                                 (F-NOR4 WRITE0 WRITE1 SEFB0 SEFB1))
                         RN-B
                         (F$V-IF-F-4 (F-OR RESET0 RESET1)
                                     (FV-IF RESET2
                                            (F$V-INC4$V REGS-ADDRESS)
                                            PC-REG))))

                 (APPEND
                  ;;ALU-OP
                  ALU-OP
                  ;;ALU-MPG
                  (F$MPG (CONS ALU-ZERO (CONS ALU-SWAP ALU-OP)))))))))))))


(disable f$cv)


(prove-lemma make-list-crock-for-f$cv=cv (rewrite)
  (implies
   (boolp v)
   (equal (make-list 4 v)
          (if v (list t t t t) (list f f f f))))
  ;;Hint
  ((enable open-make-list boolp)))


(disable make-list-crock-for-f$cv=cv)
```

```
(prove-lemma f$cv=cv (rewrite)
  (implies
   (and (bvp decoded-state) (equal (length decoded-state) 32)
        (bvp rn-a) (equal (length rn-a) 4)
        (bvp rn-b) (equal (length rn-b) 4)
        (bvp op-code) (equal (length op-code) 4)
        (bvp pc-reg) (equal (length pc-reg) 4)
        (bvp regs-address) (equal (length regs-address) 4)
        (bvp set-flags) (equal (length set-flags) 4)
        (boolp store) (boolp c) (boolp a-immediate-p) (boolp rw-)
        (boolp direct-a) (boolp side-effect-a) (boolp side-effect-b)
        (boolp pre-dec-a) (boolp pre-dec-b))
   (equal (f$cv decoded-state
                rn-a rn-b op-code
                pc-reg regs-address set-flags
                store c a-immediate-p rw-
                direct-a side-effect-a side-effect-b
                pre-dec-a pre-dec-b)
          (cv decoded-state
              rn-a rn-b op-code
              pc-reg regs-address set-flags
              store c a-immediate-p rw-
              direct-a side-effect-a side-effect-b
              pre-dec-a pre-dec-b)))
  ;;Hint
  ((enable f$cv cv boolp-b-gates if* make-list-crock-for-f$cv=cv)
   (disable if*-cons if*-c-x-x open-v-threefix)
   (disable-theory f-gates b-gates)))


(prove-lemma bvp-length-cv (rewrite)
  (let ((cv (cv decoded-state
                rn-a rn-b op-code
                pc-reg regs-address set-flags
                store c a-immediate-p rw-
                direct-a side-effect-a side-effect-b
                pre-dec-a pre-dec-b)))
    (and
     (implies
      (and (equal (length set-flags) 4)
           (equal (length rn-a) 4)
           (equal (length rn-b) 4)
           (equal (length op-code) 4))
      (equal (length cv) 40))
     (implies
      (and (bvp decoded-state)
           (equal (length decoded-state) 32))
      (bvp cv))))
  ;;Hint
  ((enable cv)))
```

```
(prove-lemma properp-length-f$cv (rewrite)
  (let ((f$cv (f$cv decoded-state
                    rn-a rn-b op-code
                    pc-reg regs-address set-flags
                    store c a-immediate-p rw-
                    direct-a side-effect-a side-effect-b
                    pre-dec-a pre-dec-b)))
    (and
     (implies
      (and (equal (length set-flags) 4)
           (equal (length rn-a) 4)
           (equal (length rn-b) 4)
           (equal (length op-code) 4))
      (equal (length f$cv) 40))
     (implies
      (and (properp decoded-state)
           (equal (length decoded-state) 32))
      (properp f$cv))))
  ;;Hint
  ((enable f$cv)
   (disable-theory f-gates)))


#.`
```

```
(DEFN CV* ()
  (LIST
   'CV
   (APPEND8 #i(DECODED-STATE 0 32)
            #i(RN-A 0 4)
            #i(RN-B 0 4)
            #i(OP-CODE 0 4)
            #i(PC-REG 0 4)
            #i(REGS-ADDRESS 0 4)
            #i(SET-FLAGS 0 4)
            '(STORE C A-IMMEDIATE-P RW- DIRECT-A SIDE-EFFECT-A SIDE-EFFECT-B
                    PRE-DEC-A PRE-DEC-B))

   (APPEND6 '(NEW-RW- STROBE- HDACK-
                      WE-REGS WE-A-REG WE-B-REG WE-I-REG
                      WE-DATA-OUT WE-ADDR-OUT WE-HOLD- WE-PC-REG
                      DATA-IN-SELECT DEC-ADDR-OUT SELECT-IMMEDIATE
                      ALU-C ALU-ZERO)
            #i(STATE 0 5)
            #i(WE-FLAGS 0 4)
            #i(NEW-REGS-ADDRESS 0 4)
            #i(ALU-OP 0 4)
            #i(ALU-MPG 0 7))

   (list
    ;;  Decoded state
    ,@(iterate for (state . value) in *control-states*
        for i from 0
        collecting `(LIST ',(unstring "G-" state) '(,state) 'ID
                          (list #i(DECODED-STATE ,i))))
    ;;  Common subexpressions
    '(g0 (store-) b-not (store))
    '(g1 (alu-zero) b-or4 (fetch0 reset0 reset1 reset2))
    '(g2 (alu-swap) b-or3 (readb2 write1 sefb1))
    '(g3 (incdeca) b-or (reada1 sefa1))
    ;;  ALU-OP
    '(g4 (s4) b-nor4 (fetch2 incdeca alu-swap alu-zero))
    '(g5 (s5) b-nand (incdeca pre-dec-a))
    '(g6 (s6) b-nand (alu-swap pre-dec-b))
    '(g7 (s7) b-nand (s5 s6))
    (list 'g8 #i(alu-op 0 4) 'select-op-code (list* 's4 's7 #i(op-code 0 4)))
    ;;G9 was optimized away.
    ;;RW-
    '(g10 (s10) b-not (rw-))
    '(g11 (s11) b-nand (s10 fetch0))
    '(g12 (s12) b-nor3 (write1 write2 write3))
    '(g13 (new-rw-) b-and (s11 s12))
    ;;STROBE-
    '(g14 (strobe-) b-nor8
          (fetch2 fetch3 reada2 reada3 readb2 readb3 write2 write3))
    ;;HDACK-
    '(g15 (hdack-) b-not (hold0))
    ;;G 16 was the old DRIVE-ADDR-OUT.
    ;;WE-REGS
```

```
'(g17 (s17) b-nand (store update))
'(g18 (s18) b-nand (side-effect-a reada1))
'(g19 (s19) b-nand3 (store- side-effect-b readb2))
'(g20 (s20) b-nand (side-effect-b write1))
'(g21 (s21) b-nor5 (fetch2 sefa1 sefb1 reset0 reset2))
'(g22 (we-regs) b-nand5 (s17 s18 s19 s20 s21))
;;WE-A-REG
'(g23 (s23) b-nand (direct-a readb1))
'(g24 (s24) b-nor6 (fetch0 rega reada0 reada3 sefa0 reset1))
'(g25 (we-a-reg) b-nand (s23 s24))
;;WE-B-REG
'(g26 (s26) b-nor4 (regb update reada2 readb0))
'(g27 (s27) b-nor4 (readb3 write0 sefb0 reset1))
'(g28 (we-b-reg) b-nand (s26 s27))
;;WE-I-REG
'(g29 (we-i-reg) b-or (fetch3 reset1))
;;WE-DATA-OUT
'(g30 (we-data-out) b-or (write0 reset0))
;;WE-ADDR-OUT
'(g31 (s31) b-nor3 (fetch0 reada0 readb0))
'(g32 (s32) b-nor (write0 reset1))
'(g33 (we-addr-out) b-nand (s31 s32))
;;WE-HOLD-
'(g34 (we-hold-) b-or3 (fetch0 hold0 reset0))
;;WE-PC-REG
'(g35 (we-pc-reg) b-or (hold0 reset0))
;;DATA-IN-SELECT
'(g36 (data-in-select) b-or3 (fetch3 reada3 readb3))
;;DEC-ADDR-OUT
'(g37 (s37) b-nand (pre-dec-a reada0))
'(g38 (s38) b-or (readb0 write0))
'(g39 (s39) b-nand (pre-dec-b s38))
'(g40 (dec-addr-out) b-nand (s37 s39))
;;SELECT-IMMEDIATE
'(g41 (s41) b-or (rega readb1))
'(g42 (select-immediate) b-and (a-immediate-p s41))
;;ALU-C
(list 'g43 '(alu-c) 'carry-in-help
      (cons 'c (cons 'alu-zero #i(alu-op 0 4))))
;;ALU-ZERO (above)
;;STATE
(list 'g44 #i(state 0 5) 'encode-32 #i(decoded-state 0 32))
;;WE-FLAGS
(list 'g45 #i(fanout-reset0 0 4) 'fanout-4 '(reset0))
'(g46 (s46) b-nor (update write0))
(list 'g47 #i(we-flags 0 4) #i(tv-if (tree-number (make-tree 4)))
      (cons 's46 (append #i(fanout-reset0 0 4) #i(set-flags 0 4))))
;;REGS-ADDRESS
'(g48 (s48) b-nor3 (rega reada0 reada1))
'(g49 (s49) b-nor3 (readb1 sefa0 sefa1))
'(g50 (select-rn-a) b-nand (s48 s49))
'(g51 (s51) b-nor5 (regb update reada2 readb0 readb2))
'(g52 (s52) b-nor4 (write0 write1 sefb0 sefb1))
'(g53 (select-rn-b) b-nand (s51 s52))
'(g54 (select-all-f) b-or (reset0 reset1))
```

```
    (list 'g55 #i(v-inc-regs-address 0 4) 'v-inc4 #i(regs-address 0 4))
    (list 'g56 #i(s56 0 4) #i(tv-if (tree-number (make-tree 4)))
          (cons 'reset2 (append #i(v-inc-regs-address 0 4) #i(pc-reg 0 4))))
    (list 'g57 #i(s57 0 4) 'v-if-f-4 (cons 'select-all-f #i(s56 0 4)))
    (list 'g58 #i(s58 0 4) #i(tv-if (tree-number (make-tree 4)))
          (cons 'select-rn-b (append #i(rn-b 0 4) #i(s57 0 4))))
    (list 'g59 #i(new-regs-address 0 4) #i(tv-if (tree-number (make-tree 4)))
          (cons 'select-rn-a (append #i(rn-a 0 4) #i(s58 0 4))))
    ;;ALU-OP (above)
    ;;ALU-MPG
    (list 'g60 #i(alu-mpg 0 7) 'mpg
          (list* 'alu-zero 'alu-swap #i(alu-op 0 4))))
   NIL))


(defn cv& (netlist)
  (and (equal (lookup-module 'cv netlist) (cv*))
       (let ((netlist (delete-module 'cv netlist)))
         (and (id& netlist)
              (b-not& netlist)
              (b-or& netlist)
              (b-or3& netlist)
              (b-or4& netlist)
              (b-nand& netlist)
              (b-nand3& netlist)
              (b-nand5& netlist)
              (b-nor& netlist)
              (b-nor3& netlist)
              (b-nor4& netlist)
              (b-nor5& netlist)
              (b-nor6& netlist)
              (b-nor8& netlist)
              (b-and& netlist)
              (select-op-code& netlist)
              (tv-if& netlist (make-tree 4))
              (carry-in-help& netlist)
              (encode-32& netlist)
              (fanout-4& netlist)
              (v-inc4& netlist)
              (v-if-f-4& netlist)
              (mpg& netlist)))))


(disable cv&)
```

```
(defn cv$netlist ()
  (cons (cv*)
        (union (union (union (union (union (id$netlist)
                                           (b-not$netlist))
                                    (union (b-or$netlist)
                                           (union (b-or3$netlist)
                                                  (b-or4$netlist))))
                             (union (union (b-nand$netlist)
                                           (b-nand3$netlist))
                                    (union (b-nor$netlist)
                                           (b-nor3$netlist))))
                      (union (union (union (b-nor4$netlist)
                                           (b-nor5$netlist))
                                    (union (b-nor6$netlist)
                                           (b-nor8$netlist)))
                             (union (union (b-and$netlist)
                                           (select-op-code$netlist))
                                    (union (tv-if$netlist (make-tree 4))
                                           (carry-in-help$netlist)))))
               (union (union (union (encode-32$netlist)
                                    (fanout-4$netlist))
                             (union (v-inc4$netlist)
                                    (v-if-f-4$netlist)))
                      (union (mpg$netlist)
                             (b-nand5$netlist))))))


(prove-lemma check-cv$netlist ()
  (cv& (cv$netlist)))
```

```
(prove-lemma cv$value (rewrite)
  (implies
   (and (cv& netlist)
        (properp decoded-state) (equal (length decoded-state) 32)
        (properp rn-a) (equal (length rn-a) 4)
        (properp rn-b) (equal (length rn-b) 4)
        (properp op-code) (equal (length op-code) 4)
        (properp pc-reg) (equal (length pc-reg) 4)
        (properp regs-address) (equal (length regs-address) 4)
        (properp set-flags) (equal (length set-flags) 4))
   (equal (dual-eval 0 'cv
                     (append
                      decoded-state
                      (append
                       rn-a
                       (append
                        rn-b
                        (append
                         op-code
                         (append
                          pc-reg
                          (append
                           regs-address
                           (append
                            set-flags
                            (list store c a-immediate-p rw- direct-a
                                  side-effect-a side-effect-b
                                  pre-dec-a pre-dec-b))))))))
                     state netlist)
          (f$cv decoded-state
                rn-a rn-b op-code
                pc-reg regs-address set-flags
                store c a-immediate-p rw-
                direct-a side-effect-a side-effect-b
                pre-dec-a pre-dec-b)))
  ;;Hint
  ((enable f$cv cv&
           id$value b-not$value b-or$value b-or3$value b-or4$value b-nand$value
           b-nand5$value b-nand3$value b-nor$value b-nor3$value b-nor4$value
           b-nor5$value b-nor6$value b-nor8$value b-and$value
           select-op-code$value tv-if$value carry-in-help$value
           encode-32$value-on-a-vector
           fanout-4$value
           v-inc4$value v-if-f-4$value mpg$value)
   (disable f-gates=b-gates open-indices *1*indices indices mpg *1*mpg
            encode-32 *1*encode-32 tree-number *1*tree-number
            make-tree *1*make-tree)
   (disable-theory f-gates)))
```

```
(disable cv$value)

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    NEXT-CNTL-STATE
;;;
;;;    The control logic.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn next-cntl-state (reset- dtack- hold- rw-
                             state i-reg flags pc-reg regs-address)
  #.(control-let
     ;;  This forces an illegal state if RESET- is asserted, thus sending the
     ;;  machine to RESET0.
     '(let ((decoded-state (decode-5 (v-if reset- state (v_v11111)))))
        (let ((next-state
               (next-state decoded-state store set-some-flags unary direct-a
                           direct-b side-effect-a side-effect-b
                           all-t-regs-address dtack- hold-)))
          (cv next-state
              rn-a rn-b op-code
              pc-reg regs-address set-flags
              store c a-immediate-p rw-
              direct-a side-effect-a side-effect-b
              pre-dec-a pre-dec-b)))))


(disable next-cntl-state)


(prove-lemma bvp-length-next-cntl-state (rewrite)
  (and
   (equal (length (next-cntl-state
                   reset- dtack- hold- rw-
                   state i-reg flags pc-reg regs-address))
          40)
   (bvp (next-cntl-state
         reset- dtack- hold- rw-
         state i-reg flags pc-reg regs-address)))
  ;;Hint
  ((enable next-cntl-state)))
```

```
(defn f$next-cntl-state (reset- dtack- hold- rw-
                              state i-reg flags pc-reg regs-address)
  (let ((control-let (f$control-let i-reg flags regs-address))
        (set-flags (set-flags i-reg))
        (op-code (op-code i-reg))
        (rn-a (rn-a i-reg))
        (rn-b (rn-b i-reg)))
    (let
      ((a-immediate-p      (car control-let))
       (store              (cadr control-let))
       (set-some-flags     (caddr control-let))
       (direct-a           (cadddr control-let))
       (direct-b           (caddddr control-let))
       (unary              (cadddddr control-let))
       (pre-dec-a          (caddddddr control-let))
       (pre-dec-b          (cadddddddr control-let))
       (c                  (caddddddddr control-let))
       (all-t-regs-address (cadddddddddr control-let))
       (side-effect-a      (caddddddddddr control-let))
       (side-effect-b      (cadddddddddddr control-let)))
      ;;  this forces an illegal state if reset- is asserted, thus sending the
      ;;  machine to reset0.
      (let ((decoded-state (f$decode-5 (fv-or
                                        (make-list 5 (f-not reset-))
                                        state))))
        (let ((next-state
               (f$next-state decoded-state store set-some-flags unary direct-a
                             direct-b side-effect-a side-effect-b
                             all-t-regs-address dtack- hold-)))
          (f$cv next-state
                rn-a rn-b op-code
                pc-reg regs-address set-flags
                store c a-immediate-p rw-
                direct-a side-effect-a side-effect-b
                pre-dec-a pre-dec-b))))))


(disable f$next-cntl-state)


(prove-lemma properp-length-f$next-cntl-state (rewrite)
  (and
   (equal (length (f$next-cntl-state
                   reset- dtack- hold- rw-
                   state i-reg flags pc-reg regs-address))
          40)
   (properp (f$next-cntl-state
             reset- dtack- hold- rw-
             state i-reg flags pc-reg regs-address)))
  ;;Hint
  ((enable f$next-cntl-state)
   (disable-theory f-gates)))
```

```
(prove-lemma v-or-crock-for-f$next-cntl-state (rewrite)
  (implies
   (and (equal (length state) 5)
        (bvp state))
   (and (equal (v-or (list f f f f f) state)
               state)
        (equal (v-or (list t t t t t) state)
               (list t t t t t))))
  ;;Hint
  ((enable v-or equal-length-add1)))


(disable v-or-crock-for-f$next-cntl-state)


(prove-lemma f$next-cntl-state=next-cntl-state (rewrite)
  (implies
   (and (boolp reset-) (boolp dtack-) (boolp hold-) (boolp rw-)
        (bvp state) (equal (length state) 5)
        (bvp i-reg) (equal (length i-reg) 32)
        (bvp flags) (equal (length flags) 4)
        (bvp pc-reg) (equal (length pc-reg) 4)
        (bvp regs-address) (equal (length regs-address) 4))
   (equal (f$next-cntl-state
             reset- dtack- hold- rw-
             state i-reg flags pc-reg regs-address)
          (next-cntl-state
             reset- dtack- hold- rw-
             state i-reg flags pc-reg regs-address)))
  ;;Hint
  ((enable f$next-cntl-state next-cntl-state boolp-b-gates
           fv-if-rewrite v_v11111 v-or-crock-for-f$next-cntl-state)
   (disable next-state)
   (enable-theory ir-fields-theory)
   (disable-theory f-gates)))
```

```
(defn next-cntl-state* ()
  (list
   'next-cntl-state
   (append6 '(reset- dtack- hold- rw-)
            #i(state 0 5)
            #i(i-reg 0 32)
            #i(flags 0 4)
            #i(pc-reg 0 4)
            #i(regs-address 0 4))
   #i(next-cntl-state 0 40)
   (list
    ;;  Decoded control lines.
    (list 'control-signals
          '(a-immediate-p store set-some-flags direct-a direct-b unary
                          pre-dec-a pre-dec-b c all-t-regs-address
                          side-effect-a side-effect-b)
          'control-let
          (append #i(i-reg 0 32) (append #i(flags 0 4) #i(regs-address 0 4))))
    ;;  Illegal state on RESET.
    (list 'not-reset
          '(reset)
          'b-not
          '(reset-))
    (list 'reset5x
          #i(reset5x 0 5)
          'fanout-5
          '(reset))
    (list 'xstate
          #i(xstate 0 5)
          #i(v-or 5)
          (append #i(reset5x 0 5) #i(state 0 5)))
    ;;  The decoded state.
    (list 'dstate
          #i(decoded-state 0 32)
          'decode-5
          #i(xstate 0 5))
    ;;  The next state
    (list 'nxstate
          #i(next-state 0 32)
          'next-state
          (append #i(decoded-state 0 32)
                  '(store set-some-flags unary direct-a direct-b
                          side-effect-a side-effect-b all-t-regs-address
                          dtack- hold-)))
    ;;  The control vector.
    (list 'cvector
          #i(next-cntl-state 0 40)
          'cv
          (append8 #i(next-state 0 32)
                   (rn-a #i(i-reg 0 32))
                   (rn-b #i(i-reg 0 32))
                   (op-code #i(i-reg 0 32))
                   #i(pc-reg 0 4)
                   #i(regs-address 0 4)
```

```
                  (set-flags #i(i-reg 0 32))
                  '(store c a-immediate-p rw- direct-a
                          side-effect-a side-effect-b
                          pre-dec-a pre-dec-b))))
   nil))
```

```
(defn next-cntl-state& (netlist)
  (and (equal (lookup-module 'next-cntl-state netlist)
              (next-cntl-state*))
       (let ((netlist (delete-module 'next-cntl-state netlist)))
         (and (control-let& netlist)
              (b-not& netlist)
              (fanout-5& netlist)
              (v-or& netlist 5)
              (decode-5& netlist)
              (next-state& netlist)
              (cv& netlist)))))
```

```
(disable next-cntl-state&)
```

```
(defn next-cntl-state$netlist ()
  (cons (next-cntl-state*)
        (union (union (union (union (control-let$netlist)
                                    (b-not$netlist))
                             (union (fanout-5$netlist)
                                    (v-or$netlist 5)))
                      (union (decode-5$netlist)
                             (next-state$netlist)))
               (cv$netlist))))
```

```
(prove-lemma check-next-cntl-state$netlist ()
  (next-cntl-state& (next-cntl-state$netlist)))
```

```
(prove-lemma next-cntl-state$value (rewrite)
  (implies
   (and (next-cntl-state& netlist)
        (properp state) (equal (length state) 5)
        (properp i-reg) (equal (length i-reg) 32)
        (properp flags) (equal (length flags) 4)
        (properp pc-reg) (equal (length pc-reg) 4)
        (properp regs-address) (equal (length regs-address) 4))
   (equal (dual-eval 0 'next-cntl-state
                     (list* reset- dtack- hold- rw-
                            (append
                             state
                             (append
                              i-reg
                              (append flags (append pc-reg regs-address)))))
                     dual-eval-state netlist)
          (f$next-cntl-state reset- dtack- hold- rw-
                             state i-reg flags pc-reg regs-address)))
  ;;Hint
  ((enable next-cntl-state& f$next-cntl-state
           control-let$value b-not$value fanout-5$value v-or$value
           decode-5$value next-state$value cv$value
           c-flag v_v11111)
   (enable-theory ir-fields-theory)
   (disable indices *1*indices open-indices decode-5 *1*decode-5
            cv *1*cv f$next-state *1*f$next-state)))


(disable next-cntl-state$value)

;;;  This macro generates that lemmas that prove that the control logic
;;;  produces the desired control vectors.


(generate-next-cntl-state-lemmas)
```

14.59 "regfile.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    REGFILE.EVENTS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    Boolean specifications of register file access.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn read-regs (address regs)
  (let ((regfile      (car regs))
        (last-we      (cadr regs))
        (last-data    (caddr regs))
        (last-address (cadddr regs)))
    (if (and last-we (v-iff address last-address))
        last-data
      (read-mem address regfile))))


(disable read-regs)


(defn write-regs (we address regs data)
  (let ((regfile      (car regs))
        (last-we      (cadr regs))
        (last-data    (caddr regs))
        (last-address (cadddr regs)))
    (list (if last-we (write-mem last-address regfile last-data) regfile)
          we
          (if we data last-data)
          (if we address last-address))))


(disable write-regs)


(prove-lemma write-regs-f-write-regs-f (rewrite)
  (equal (write-regs f addr1 (write-regs f addr2 regs data2) data1)
         (write-regs f addr1 regs data1))
  ;;Hint
  ((enable write-regs)))
```

```
(prove-lemma write-regs-f-bv-crock (rewrite)
  (equal (write-regs f addr regs (bv x))
         (write-regs f 0 regs 0))
  ;;Hint
  ((enable write-regs)))


(prove-lemma properp-length-write-regs (rewrite)
  (and (properp (write-regs we address regs data))
       (equal (length (write-regs we address regs data)) 4))
  ;;Hint
  ((enable write-regs)))


(prove-lemma write-regs-if (rewrite)
  (equal (write-regs (if c a b) address regs data)
         (if c
             (write-regs a address regs data)
           (write-regs b address regs data))))


(prove-lemma write-regs-ok (rewrite)
  (and
   (equal (memory-okp n m (car (write-regs we address regs data)))
          (if (cadr regs)
              (memory-okp n m (write-mem (cadddr regs)
                                         (car regs)
                                         (caddr regs)))
            (memory-okp n m (car regs))))
   (equal (boolp (cadr (write-regs we address regs data)))
          (boolp we))
   (equal (bvp (caddr (write-regs we address regs data)))
          (if we
              (bvp data)
            (bvp (caddr regs))))
   (equal (length (caddr (write-regs we address regs data)))
          (if we
              (length data)
            (length (caddr regs))))
   (equal (bvp (cadddr (write-regs we address regs data)))
          (if we
              (bvp address)
            (bvp (cadddr regs))))
   (equal (length (cadddr (write-regs we address regs data)))
          (if we
              (length address)
            (length (cadddr regs)))))
  ;;Hint
  ((enable write-regs)))
```

```
(prove-lemma read-regs-write-regs-f (rewrite)
  (implies
   (and (all-ramp-mem (length addr) (car regs))
        (equal (length addr) (length (cadddr regs)))
        (bvp addr))
   (equal (read-regs addr (write-regs f address regs value))
          (read-regs addr regs)))
   ;;hint
   ((enable read-regs write-regs)
    (use (read-mem-write-mem (v-addr1 addr)
                             (v-addr2 (cadddr regs))
                             (value (caddr regs))
                             (mem (car regs))))))


(prove-lemma all-ramp-mem-after-write-regs (rewrite)
  (implies
   (and (all-ramp-mem n (car regs))
        (equal (length (cadddr regs)) n))
   (all-ramp-mem n (car (write-regs we address regs data))))
  ;;Hint
  ((enable write-regs)))


(prove-lemma read-regs=read-mem (rewrite)
  (implies
   (not (cadr regs))
   (equal (read-regs v-addr regs)
          (read-mem v-addr (car regs))))
  ;;Hint
  ((enable read-regs)))


(prove-lemma read-regs=read-mem-write-mem (rewrite)
  (implies
   (and (all-ramp-mem (length v-addr1) (car regs))
        (equal (length v-addr1) (length (cadddr regs)))
        (cadr regs))
   (equal (read-regs v-addr1 regs)
          (read-mem v-addr1
                    (write-mem (cadddr regs) (car regs) (caddr regs)))))
  ;;Hint
  ((use (read-mem-write-mem (mem (car regs))
                            (v-addr2 (cadddr regs))
                            (value (caddr regs))))
   (enable read-regs)))
```

```
(prove-lemma bvp-length-read-regs-32 (rewrite)
  (implies
   (and (memory-okp (length v-addr) 32 (car regs))
        (bvp (caddr regs))
        (equal (length (caddr regs)) 32))
   (and (bvp (read-regs v-addr regs))
        (equal (length (read-regs v-addr regs))
               32)))
  ;;Hint
  ((enable read-regs)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    Hardware Implementation of the Register File
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(module-generator
 (regfile*)
 'REGFILE
 (list* 'clk 'te 'ti 'we 'disable-regfile- 'test-regfile-
        (append #i(address 0 4) #i(data 0 32)))
 (cons 'to #i(out 0 32))
 (list
  ;;  WE-LATCH. Scan TI to WE-DP-RAM
  (list 'we-latch
        '(we-dp-ram we-dp-ram-)
        'fd1s
        '(we clk ti te))
  ;;  ADDRESS-LATCH.  Scan WE-DP-RAM to #i(address-dp-ram 3)
  (list 'address-latch
        #i(address-dp-ram 0 4)
        #i(we-reg 4)
        (list* 'clk 'we 'te 'we-dp-ram #i(address 0 4)))
  ;;  DATA-LATCH.  Scan #i(address-dp-ram 3) to #i(data-dp-ram 31)
  (list 'data-latch
        #i(data-dp-ram 0 32)
        #i(we-reg 32)
        (list* 'clk 'we 'te #i(address-dp-ram 3) #i(data 0 32)))

  ;; Register File Enable Circuit
  '(reg-en-circuit (we-) ram-enable-circuit
                   (clk test-regfile- disable-regfile- we-dp-ram))
  ;;  The RAM.  This is a level-sensitive device.  The surrounding circuitry
  ;;  makes the entire register file work as if it were an edge-triggered
  ;;  device.
  (list 'ram
        #i(ramout 0 32)
        'dp-ram-16x32
        (append #i(address 0 4)                        ;Read address
                (append #i(address-dp-ram 0 4)         ;Write address
                        (cons 'we-                     ;Write enable
                              #i(data-dp-ram 0 32))))) ;Data
  ;;  Address comparator.
  (list 'compare
        '(read-equal-write)
        #i(v-equal 4)
        (append #i(address 0 4) #i(address-dp-ram 0 4)))
  ;;  Mux control
  (list 'mux-control
        '(s)
        'b-and3
        '(we-dp-ram read-equal-write test-regfile-))
  ;;  Mux
  (list 'mux
        #i(out 0 32)
        #i(tv-if (tree-number (make-tree 32)))
        (cons 's (append #i(data-dp-ram 0 32) #i(ramout 0 32))))
  ;;  Rename the scan out
  (list 'scanout
        '(to)
```

```
        'id
        (list #i(data-dp-ram 31))))
 '(ram we-latch data-latch address-latch))


(disable *1*regfile*)


(defn regfile& (netlist)
  (and (equal (lookup-module 'regfile netlist) (regfile*))
       (let ((netlist (delete-module 'regfile netlist)))
         (and (fd1s& netlist)
              (we-reg& netlist 4)
              (we-reg& netlist 32)
              (ram-enable-circuit& netlist)
              (dp-ram-16x32& netlist)
              (v-equal& netlist 4)
              (b-and3& netlist)
              (tv-if& netlist (make-tree 32))
              (id& netlist)))))


(disable regfile&)


(defn regfile$netlist ()
  (cons (regfile*)
        (union (union (union (union (fd1s$netlist)
                                    (we-reg$netlist 4))
                             (union (we-reg$netlist 32)
                                    (ram-enable-circuit$netlist)))
                      (union (dp-ram-16x32$netlist)
                             (union (v-equal$netlist 4)
                                    (b-and3$netlist))))
               (union (tv-if$netlist (make-tree 32))
                      (id$netlist)))))


(prove-lemma check-regfile$netlist ()
  (regfile& (regfile$netlist))
  ;;Hint
  ((expand (regfile*))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    4-Valued Register File Specifications
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;   F$READ-REGS
```

```
(defn f$read-regs (address regs)
  (let ((regfile      (car regs))
        (last-we      (cadr regs))
        (last-data    (caddr regs))
        (last-address (cadddr regs)))
    (fv-if
     (f-and3 (f-buf last-we)
             (f$v-equal address (v-threefix last-address))
             t)
     (v-threefix last-data)
     (dual-port-ram-value
      32 4
      (append address
              (append (v-threefix last-address)
                      (cons (f-nand t (f-buf last-we))
                            (v-threefix last-data))))
      regfile))))


(disable f$read-regs)


(prove-lemma properp-length-f$read-regs (rewrite)
  (and (properp (f$read-regs address mem))
       (implies
        (equal (length (caddr mem)) 32)
        (equal (length (f$read-regs address mem)) 32)))
  ;;Hint
  ((enable f$read-regs)))


(prove-lemma f$read-regs=read-regs (rewrite)
  (implies
   (and (memory-okp 4 32 (car regs))
        (boolp (cadr regs))
        (bvp (caddr regs)) (equal (length (caddr regs)) 32)
        (bvp (cadddr regs)) (equal (length (cadddr regs)) 4)
        (bvp address) (equal (length address) 4))
   (equal (f$read-regs address regs)
          (read-regs address regs)))
  ;;Hint
  ((enable read-regs f$read-regs boolp-b-gates open-nth)
   (disable *1*make-list)
   (disable-theory f-gates)))

;;  The $VALUE lemma is a little more general than the $STATE lemma.
```

```
(prove-lemma regfile$value (rewrite)
  (implies
   (and (regfile& netlist)
        (equal test-regfile- t)
        (equal disable-regfile- t)
        (memory-properp 4 32 mem)
        (properp last-data) (equal (length last-data) 32)
        (properp address) (equal (length address) 4)
        (properp last-address) (equal (length last-address) 4))
   (equal (cdr (dual-eval 0 'regfile
                          (list* clk te ti we disable-regfile- test-regfile-
                                 (append address data))
                          (list mem last-we last-data last-address)
                          netlist))
          (f$read-regs address (list mem last-we last-data last-address))))
  ;;Hint
  ((enable regfile& f$read-regs regfile*$destructure open-nth
           fd1s$value we-reg$value ram-enable-circuit$value
           dp-ram-16x32$structured-value
           v-equal$value b-and3$value tv-if$value id$value)
   (disable indices *1*indices open-indices open-v-threefix
            make-tree *1*make-tree make-list *1*make-list
            threefix dual-port-ram-value)
   (disable-theory f-gates)))


(disable regfile$value)


(defn f$write-regs (we address regs data)
  (let ((regfile      (car regs))
        (last-we      (cadr regs))
        (last-data    (caddr regs))
        (last-address (cadddr regs)))
    (list
     (dual-port-ram-state
      32 4
      (append address
              (append (v-threefix last-address)
                      (cons (f-nand t (f-buf last-we))
                            (v-threefix last-data))))
      regfile)
     (threefix we)
     (fv-if we
            data
            last-data)
     (fv-if we
            address
            last-address))))


(disable f$write-regs)
```

```
(prove-lemma properp-length-f$write-regs (rewrite)
  (and (properp (f$write-regs we address regs data))
       (equal (length (f$write-regs we address regs data))
              4))
  ;;Hint
  ((enable f$write-regs)
   (disable dual-port-ram-state threefix)))


(prove-lemma f$write-regs=write-regs (rewrite)
  (implies
   (and
    (boolp we)
    (boolp (cadr regs))
    (bvp (caddr regs)) (equal (length (caddr regs)) 32)
    (bvp (cadddr regs)) (equal (length (cadddr regs)) 4)
    (bvp address) (equal (length address) 4)
    (bvp data) (equal (length data) 32))
   (equal (f$write-regs we address regs data)
          (write-regs we address regs data)))
  ;;Hint
  ((enable f$write-regs write-regs boolp-b-gates open-nth subrange-cons)
   (disable-theory f-gates)))
```

```
(prove-lemma regfile$state (rewrite)
  (implies
   (and (regfile& netlist)
        (not te)
        (equal test-regfile- t)
        (equal disable-regfile- t)
        (properp data) (equal (length data) 32)
        (properp last-data) (equal (length last-data) 32)
        (properp address) (equal (length address) 4)
        (properp last-address) (equal (length last-address) 4))
   (equal (dual-eval 2 'regfile
                     (list* clk te ti we disable-regfile- test-regfile-
                            (append address data))
                     (list regfile last-we last-data last-address)
                     netlist)
          (f$write-regs we address
                        (list regfile last-we last-data last-address)
                        data)))
  ;;Hint
  ((enable regfile& f$write-regs regfile*$destructure open-nth
           fd1s$state fd1s$value we-reg$state we-reg$value
           ram-enable-circuit$value
           dp-ram-16x32$structured-state
           v-equal$value b-and3$value tv-if$value id$value)
   (disable indices *1*indices open-indices open-v-threefix
            make-tree *1*make-tree make-list *1*make-list
            threefix dual-port-ram-state)
   (expand (f-if f ti we) (f-buf we))
   (disable-theory f-gates)))


(disable regfile$state)
```

14.60 "flags.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    FLAGS.EVENTS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  The FLAGS register is simply four, individually write-enabled, 1-bit
;;;  latches.  For simplicity the FLAGS register takes the entire 35-bit ALU
;;;  bus as an input, even though it only needs 4 of the bits, i.e., the C, V,
;;;  N, and Z.  Test input = TI.  Test output = C.


(module-generator
 (flags*)
 'flags
 (list* 'clk 'te 'ti (append #i(set-flags 0 4) #i(cvzbv 0 35)))
 '(z n v c)
 (list
  ;; Z. Scan TI to Z
  (list 'z-latch
        '(z zb)
        'fd1slp
        (list (zb #i(cvzbv 0 35)) 'clk (z-set #i(set-flags 0 4)) 'ti 'te))
  ;; N. Scan Z to N
  (list 'n-latch
        '(n nb)
        'fd1slp
        (list #i(cvzbv 34)        'clk (n-set #i(set-flags 0 4)) 'z  'te))
  ;; V. Scan N to V
  (list 'v-latch
        '(v vb)
        'fd1slp
        (list (v #i(cvzbv 0 35))  'clk (v-set #i(set-flags 0 4)) 'n  'te))
  ;; C. Scan V to C
  (list 'c-latch
        '(c cb)
        'fd1slp
        (list (c #i(cvzbv 0 35))  'clk (c-set #i(set-flags 0 4)) 'v  'te)))
 '(z-latch n-latch v-latch c-latch))


(disable *1*flags*)
```

```
(defn flags& (netlist)
  (and (equal (lookup-module 'flags netlist)
              (flags*))
       (let ((netlist (delete-module 'flags netlist)))
         (fd1slp& netlist))))


(disable flags&)


(defn flags$netlist ()
  (cons (flags*) (fd1slp$netlist)))


(prove-lemma check-flags$netlist ()
  (flags& (flags$netlist))
  ;;Hint
  ((expand (flags*))))


(prove-lemma flags$value (rewrite)
  (implies
   (and (flags& netlist)
        (properp flags) (equal (length flags) 4))
   (equal (dual-eval 0 'flags (list* clk te ti (append set-flags cvzbv))
                     flags netlist)
          (v-threefix flags)))
  ;;Hint
  ((enable flags& flags*$destructure fd1slp$value
           c-set v-set n-set z-set
           c v zb
           open-nth equal-length-add1)
   (disable indices *1*indices open-indices)))


(disable flags$value)


(defn f$update-flags (flags set-flags cvzbv)
  (list (f-if (z-set set-flags)
              (zb cvzbv)
              (z-flag flags))
        (f-if (n-set set-flags)
              (n cvzbv)
              (n-flag flags))
        (f-if (v-set set-flags)
              (v cvzbv)
              (v-flag flags))
        (f-if (c-set set-flags)
              (c cvzbv)
              (c-flag flags))))
```

```
(disable f$update-flags)


(prove-lemma properp-length-f$update-flags (rewrite)
  (and (properp (f$update-flags flags set-flags cvzbv))
       (equal (length (f$update-flags flags set-flags cvzbv)) 4))
  ;;Hint
  ((enable f$update-flags)))


(prove-lemma f$update-flags=update-flags (rewrite)
  (implies
   (and
    (bvp flags) (equal (length flags) 4)
    (bvp set-flags) (equal (length set-flags) 4)
    (bvp cvzbv) (equal (length cvzbv) 35))
   (equal (f$update-flags flags set-flags cvzbv)
          (update-flags flags set-flags cvzbv)))
  ;;Hint
  ((enable f$update-flags update-flags
           z-set zb z-flag n-set n n-flag v-set v v-flag c-set c c-flag)
   (disable-theory f-gates)))


(prove-lemma flags$state-help (rewrite)
  (implies
   (and (flags& netlist)
        (properp flags) (equal (length flags) 4)
        (equal set-flags (list set-z set-n set-v set-c))
        (equal cvzbv (cons c (cons v (cons z bv))))
        (properp cvzbv) (equal (length cvzbv) 35)
        (not te))
   (equal (dual-eval 2 'flags (list* clk te ti (append set-flags cvzbv))
                     flags netlist)
          (f$update-flags flags set-flags cvzbv)))
  ;;Hint
  ((enable flags& f$update-flags flags*$destructure fd1slp$state
           c-set v-set n-set z-set
           c-flag v-flag n-flag z-flag
           c v zb n bv
           update-flags nth v-negp-as-nth)
   (disable indices *1*indices open-indices)
   (expand #i(set-flags 0 4)
           #i(set-flags 1 3)
           #i(set-flags 2 2)
           #i(set-flags 3 1)
           #i(cvzbv 0 35)
           #i(cvzbv 1 34)
           #i(cvzbv 2 33))))


(disable flags$state-help)
```

```
(prove-lemma flags$state (rewrite)
  (implies
   (and (flags& netlist)
        (properp flags) (equal (length flags) 4)
        (properp set-flags) (equal (length set-flags) 4)
        (properp cvzbv) (equal (length cvzbv) 35)
        (not te))
   (equal (dual-eval 2 'flags (list* clk te ti (append set-flags cvzbv))
                     flags netlist)
          (f$update-flags flags set-flags cvzbv)))
  ;;Hint
  ((use (flags$state-help
         (set-z (car set-flags))
         (set-n (cadr set-flags))
         (set-v (caddr set-flags))
         (set-c (cadddr set-flags))
         (c (car cvzbv))
         (v (cadr cvzbv))
         (z (caddr cvzbv))
         (bv (cdddr cvzbv))))))


(disable flags$state)


(prove-lemma flags$partial-state-help (rewrite)
  (implies
   (and (flags& netlist)
        (properp flags) (equal (length flags) 4)
        (equal set-flags (make-list 4 t))
        (equal cvzbv (cons c (cons v (cons z bv))))
        (bvp cvzbv) (equal (length cvzbv) 35)
        (not te))
   (equal (dual-eval 2 'flags (list* clk te ti (append set-flags cvzbv))
                     flags netlist)
          (update-flags flags set-flags cvzbv)))
  ;;Hint
  ((enable flags& flags*$destructure fd1slp$state
           c-set v-set n-set z-set
           c-flag v-flag n-flag z-flag
           c v zb n bv
           update-flags nth v-negp-as-nth)
   (disable indices *1*indices open-indices)
   (expand #i(set-flags 0 4)
           #i(set-flags 1 3)
           #i(set-flags 2 2)
           #i(set-flags 3 1)
           #i(cvzbv 0 35)
           #i(cvzbv 1 34)
           #i(cvzbv 2 33))))


(disable flags$partial-state-help)
```

```
(prove-lemma flags$partial-state (rewrite)
  (implies
   (and (flags& netlist)
        (properp flags) (equal (length flags) 4)
        (equal set-flags (make-list 4 t))
        (bvp cvzbv) (equal (length cvzbv) 35)
        (not te))
   (equal (dual-eval 2 'flags (list* clk te ti (append set-flags cvzbv))
                     flags netlist)
          (update-flags flags set-flags cvzbv)))
  ;;Hint
  ((use (flags$partial-state-help
         (c (car cvzbv))
         (v (cadr cvzbv))
         (z (caddr cvzbv))
         (bv (cdddr cvzbv))))
   (disable *1*make-list)))


(disable flags$partial-state)
```

14.61 "extend-immediate.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    EXTEND-IMMEDIATE.EVENTS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  This module selects either a 32-bit value, or a 32-bit value produced by
;;;  sign-extending a 9-bit value into 32 bits.


(module-generator
 (extend-immediate*)
 'extend-immediate
 (cons 'select-immediate (append #i(immediate 0 9) #i(reg-data 0 32)))
 #i(z 0 32)
 (list
  (list 'buffer
        '(sign-bit)
        'b-buf-pwr
        (list #i(immediate 8)))
  (list 'mux
        #i(z 0 32)
        #i(tv-if (tree-number (make-tree 32)))
        (cons 'select-immediate
              (append (append #i(immediate 0 9)
                              (make-list 23 'sign-bit))
                      #i(reg-data 0 32)))))
 nil)


(disable *1*extend-immediate*)


(defn extend-immediate& (netlist)
  (and (equal (lookup-module 'extend-immediate netlist)
              (extend-immediate*))
       (let ((netlist (delete-module 'extend-immediate netlist)))
         (and (b-buf-pwr& netlist)
              (tv-if& netlist (make-tree 32))))))


(disable extend-immediate&)


(defn extend-immediate$netlist ()
  (cons (extend-immediate*)
        (union (b-buf-pwr$netlist)
               (tv-if$netlist (make-tree 32)))))
```

```
(prove-lemma check-extend-immediate$netlist ()
  (extend-immediate& (extend-immediate$netlist))
  ;;Hint
  ((expand (extend-immediate*))))


(defn f$extend-immediate (select-immediate immediate reg-data)
  (fv-if select-immediate
         (append immediate
                 (if (boolp (nth 8 immediate))
                     (make-list 23 (nth 8 immediate))
                   (make-list 23 (x))))
         reg-data))


(disable f$extend-immediate)


(prove-lemma properp-length-f$extend-immediate (rewrite)
  (and (properp (f$extend-immediate select-immediate immediate reg-bus))
       (implies
        (equal (length immediate) 9)
       (equal (length (f$extend-immediate select-immediate immediate reg-bus))
              32)))
  ;;Hint
  ((enable f$extend-immediate)))


(prove-lemma f$extend-immediate=extend-immediate (rewrite)
  (implies
   (and (bvp immediate) (equal (length immediate) 9)
        (bvp reg-data) (equal (length reg-data) 32)
        (boolp select-immediate))
   (equal (f$extend-immediate select-immediate immediate reg-data)
          (if* select-immediate
              (sign-extend immediate 32)
            reg-data)))
  ;;Hint
  ((enable f$extend-immediate sign-extend-as-append if*)
   (disable associativity-of-append
            make-list *1*make-list
            indices *1*indices open-indices
            make-tree *1*make-tree)))
```

```
(prove-lemma extend-immediate$value (rewrite)
  (implies
   (and (extend-immediate& netlist)
        (properp immediate) (equal (length immediate) 9)
        (properp reg-data) (equal (length reg-data) 32))
   (equal (dual-eval 0 'extend-immediate
                     (cons select-immediate (append immediate reg-data))
                     state netlist)
          (f$extend-immediate select-immediate immediate reg-data)))
  ;;Hint
  ((enable f$extend-immediate extend-immediate& extend-immediate*$destructure
           b-buf-pwr$value tv-if$value)
   (disable associativity-of-append
            make-list *1*make-list
            indices *1*indices open-indices
            make-tree *1*make-tree)))


(disable extend-immediate$value)
```

14.62 "pad-vectors.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    PAD-VECTORS.EVENTS
;;;
;;;    Vector modules of I/O pads.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  It greatly simplifies the proofs at the CHIP level to package the pads
;;;  into the modules defined in this file.  However, LSI logic requires
;;;  every module to contain at least 1 "interior gate", i.e., a gate that is
;;;  not a pad.  Therefore, in some places we introduce redundant buffers.

;;;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    TTL-INPUT-PADS
;;;
;;;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn ttl-input-pads-body (m n pi)
  (if (zerop n)
      nil
    (cons
     (list #i(g m)                              ; Occurrence name
           (list #i(out m)                      ; Outputs
                 #i(po m))
           'TTL-INPUT                           ; Type TTL-INPUT
           (list #i(in m) pi))                  ; Inputs
     (cons
      (list #i(b m)             ; Useless buffers for LSI's "lpace".
            (list #i(b-out m))
            'B-BUF
            (list #i(out m)))
      (ttl-input-pads-body (add1 m) (sub1 n) #i(po m))))))


(disable ttl-input-pads-body)
```

```
(defn ttl-input-pads-body$induction
  (m n pi bindings state-bindings netlist)
  (if (zerop n)
      t
    (ttl-input-pads-body$induction
     (add1 m)
     (sub1 n)
     #i(po m)
     (dual-eval-body-bindings 2
                              (ttl-input-pads-body m
                                                   n
                                                   pi)
                              bindings
                              state-bindings
                              netlist)
     state-bindings netlist)))
```

```
(defn ttl-input-pads* (n)
  (list #i(ttl-input-pads n)
        (cons 'pi #i(in 0 n))
        (cons #i(po (sub1 n)) #i(b-out 0 n))
        (ttl-input-pads-body 0 n 'pi)
        nil))
```

```
(destructuring-lemma ttl-input-pads*)
```

```
(defn ttl-input-pads& (netlist n)
    (and (equal (lookup-module #i(ttl-input-pads n) netlist)
              (ttl-input-pads* n))
       (let ((netlist (delete-module (index 'ttl-input-pads n) netlist)))
         (and (ttl-input& netlist)
              (b-buf& netlist)))))
```

```
(disable ttl-input-pads&)
```

```
(defn ttl-input-pads$netlist (n)
  (cons (ttl-input-pads* n)
        (union (ttl-input$netlist)
               (b-buf$netlist))))
```

```
(prove-lemma ttl-input-pads-body$unbound-in-body (rewrite)
  (implies
   (lessp l m)
   (unbound-in-body #i(b-out l)
                    (ttl-input-pads-body m n pi)))
  ;;Hint
  ((enable unbound-in-body ttl-input-pads-body)))
```

```
(disable ttl-input-pads-body$unbound-in-body)


(prove-lemma ttl-input-pads-body$value (rewrite)
  (implies
   (and (ttl-input& netlist)
        (b-buf& netlist))
   (equal (collect-value
           #i(b-out m n)
           (dual-eval 1 (ttl-input-pads-body m n pi)
                      bindings state-bindings netlist))
          (v-threefix (collect-value #i(in m n) bindings))))
  ;;Hint
  ((enable ttl-input-pads-body ttl-input-pads-body$unbound-in-body
           v-threefix ttl-input$value b-buf$value)
   (induct (ttl-input-pads-body$induction m n pi
                                          bindings state-bindings netlist))))


(disable ttl-input-pads-body$value)


(prove-lemma ttl-input-pads$value (rewrite)
  (implies
   (and (ttl-input-pads& netlist n)
        (equal (length inputs) n)
        (properp inputs))
   (equal (cdr
           (dual-eval 0 #i(ttl-input-pads n) (cons pi inputs) state netlist))
          (v-threefix inputs)))
  ;;Hint
  ((enable ttl-input-pads& ttl-input-pads-body$value
           ttl-input-pads*$destructure)
   (disable v-threefix open-indices)))


(disable ttl-input-pads$value)

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    TTL-OUTPUT-PADS
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defn ttl-output-pads-body (m n)
  (if (zerop n)
      nil
    (cons
     (list #i(b m)                      ; Useless buffers for LSI's "lpace".
           (list #i(b-in m))
           'B-BUF
           (list #i(in m)))
     (cons
      (list #i(g m)                     ; Occurrence name
            (list #i(out m))            ; Outputs
            'TTL-OUTPUT                 ; Type TTL-OUTUT
            (list #i(b-in m)))          ; Inputs
      (ttl-output-pads-body (add1 m) (sub1 n))))))


(disable ttl-output-pads-body)


(defn ttl-output-pads-body$induction
  (m n bindings state-bindings netlist)
  (if (zerop n)
      t
    (ttl-output-pads-body$induction
     (add1 m)
     (sub1 n)
     (dual-eval-body-bindings 2
                              (ttl-output-pads-body m n)
                              bindings
                              state-bindings
                              netlist)
     state-bindings netlist)))


(defn ttl-output-pads* (n)
  (list #i(ttl-output-pads n)
        #i(in 0 n)
        #i(out 0 n)
        (ttl-output-pads-body 0 n)
        nil))


(destructuring-lemma ttl-output-pads*)


(defn ttl-output-pads& (netlist n)
    (and (equal (lookup-module #i(ttl-output-pads n) netlist)
                (ttl-output-pads* n))
       (let ((netlist (delete-module (index 'ttl-output-pads n) netlist)))
         (and (ttl-output& netlist)
              (b-buf& netlist)))))
```

```
(disable ttl-output-pads&)


(defn ttl-output-pads$netlist (n)
  (cons (ttl-output-pads* n)
        (union (ttl-output$netlist)
               (b-buf$netlist))))


(prove-lemma ttl-output-pads-body$unbound-in-body (rewrite)
  (implies
   (lessp l m)
   (unbound-in-body #i(out l)
                    (ttl-output-pads-body m n)))
  ;;Hint
  ((enable unbound-in-body ttl-output-pads-body)))


(disable ttl-output-pads-body$unbound-in-body)


(prove-lemma ttl-output-pads-body$value (rewrite)
  (implies
   (and (ttl-output& netlist)
        (b-buf& netlist))
   (equal (collect-value
           #i(out m n)
           (dual-eval 1 (ttl-output-pads-body m n)
                      bindings state-bindings netlist))
          (v-threefix (collect-value #i(in m n) bindings))))
  ;;Hint
  ((enable ttl-output-pads-body ttl-output-pads-body$unbound-in-body
           v-threefix ttl-output$value b-buf$value)
   (induct (ttl-output-pads-body$induction m n
                                           bindings state-bindings netlist))))


(disable ttl-output-pads-body$value)
```

```
(prove-lemma ttl-output-pads$value (rewrite)
  (implies
   (and (ttl-output-pads& netlist n)
        (equal (length inputs) n)
        (properp inputs))
   (equal (dual-eval 0 #i(ttl-output-pads n)
                     inputs state netlist)
          (v-threefix inputs)))
  ;;Hint
  ((enable ttl-output-pads& ttl-output-pads*$destructure
           ttl-output-pads-body$value)
   (disable v-threefix open-indices)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    TTL-TRI-OUTPUT-PADS
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn ttl-tri-output-pads-body (m n)
  (if (zerop n)
      nil
    (cons
     (list #i(g m)                      ; Occurrence name
           (list #i(out m))             ; Outputs
           'TTL-TRI-OUTPUT              ; Type TTL-TRI-OUTPUT
           (list #i(in m) 'enable-buf)) ; Inputs
     (ttl-tri-output-pads-body (add1 m) (sub1 n)))))


(disable ttl-tri-output-pads-body)


(generate-body-induction-scheme ttl-tri-output-pads-body)


(defn ttl-tri-output-pads* (n)
  (list #i(ttl-tri-output-pads n)
        (cons 'enable #i(in 0 n))
        #i(out 0 n)
        (cons (list 'enable-buffer '(enable-buf)
                    (if (lessp n 8) 'B-BUF 'B-BUF-PWR)
                    '(enable))
              (ttl-tri-output-pads-body 0 n))
        nil))


(destructuring-lemma ttl-tri-output-pads*)
```

```
(defn ttl-tri-output-pads& (netlist n)
    (and (equal (lookup-module #i(ttl-tri-output-pads n) netlist)
              (ttl-tri-output-pads* n))
       (let ((netlist (delete-module (index 'ttl-tri-output-pads n) netlist)))
         (and (ttl-tri-output& netlist)
              (if (lessp n 8)
                  (b-buf& netlist)
                  (b-buf-pwr& netlist))))))


(disable ttl-tri-output-pads&)


(defn ttl-tri-output-pads$netlist (n)
  (cons (ttl-tri-output-pads* n)
        (union (if (lessp n 8)
                   (b-buf$netlist)
                   (b-buf-pwr$netlist))
               (ttl-tri-output$netlist))))


(prove-lemma ttl-tri-output-pads-body$unbound-in-body (rewrite)
  (implies
   (lessp l m)
   (unbound-in-body #i(out l)
                    (ttl-tri-output-pads-body m n)))
  ;;Hint
  ((enable unbound-in-body ttl-tri-output-pads-body)))


(disable ttl-tri-output-pads-body$unbound-in-body)


(prove-lemma ttl-tri-output-pads-body$value (rewrite)
  (implies
   (ttl-tri-output& netlist)
   (equal (collect-value
           #i(out m n)
           (dual-eval 1 (ttl-tri-output-pads-body m n)
                      bindings state-bindings netlist))
          (vft-buf (f-not (value 'enable-buf bindings))
                   (collect-value #i(in m n) bindings))))
  ;;Hint
  ((enable ttl-tri-output-pads-body
           ttl-tri-output-pads-body$unbound-in-body
           v-threefix ttl-tri-output$value make-list
           vft-buf-rewrite)
   (induct
    (ttl-tri-output-pads-body$induction m n
                                        bindings state-bindings netlist))))


(disable ttl-tri-output-pads-body$value)
```

```
(prove-lemma ttl-tri-output-pads$value (rewrite)
  (implies
   (and (ttl-tri-output-pads& netlist n)
        (equal (length inputs) n)
        (properp inputs))
   (equal (dual-eval 0 #i(ttl-tri-output-pads n)
                     (cons enable inputs)
                     state netlist)
          (vft-buf (f-not enable) inputs)))
  ;;Hint
  ((enable ttl-tri-output-pads&
           ttl-tri-output-pads*$destructure
           ttl-tri-output-pads-body$value
           b-buf$value b-buf-pwr$value)
   (disable v-threefix open-indices)))


(disable ttl-tri-output-pads$value)

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    TTL-BIDIRECT-PADS
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn ttl-bidirect-pads-body (m n pi)
  (if (zerop n)
      nil
    (cons
     (list #i(g m)                              ; Occurrence name
           (list #i(data m)                     ; Outputs
                 #i(out m)
                 #i(po m))
           'TTL-BIDIRECT                        ; Type TTL-OUTUT
           (list #i(data m)                     ; Inputs
                 #i(in m)
                 'buf-enable pi))
     (ttl-bidirect-pads-body (add1 m) (sub1 n) #i(po m)))))


(disable ttl-bidirect-pads-body)


(generate-body-induction-scheme ttl-bidirect-pads-body)
```

```
(defn ttl-bidirect-pads* (n)
  (list #i(ttl-bidirect-pads n)
        (list* 'enable 'pi (append #i(data 0 n) #i(in 0 n)))
        (list* #i(po (sub1 n)) (append #i(data 0 n) #i(out 0 n)))
        (cons (list 'enable-buf '(buf-enable)
                    (if (lessp n 8) 'B-BUF 'B-BUF-PWR)
                    '(enable))
              (ttl-bidirect-pads-body 0 n 'pi))
        nil))


(destructuring-lemma ttl-bidirect-pads*)


(defn ttl-bidirect-pads& (netlist n)
    (and (equal (lookup-module #i(ttl-bidirect-pads n) netlist)
                (ttl-bidirect-pads* n))
       (let ((netlist (delete-module (index 'ttl-bidirect-pads n) netlist)))
         (and (ttl-bidirect& netlist)
              (if (lessp n 8)
                  (b-buf& netlist)
                  (b-buf-pwr& netlist))))))


(disable ttl-bidirect-pads&)


(defn ttl-bidirect-pads$netlist (n)
  (cons (ttl-bidirect-pads* n)
        (union (ttl-bidirect$netlist)
               (if (lessp n 8)
                   (b-buf$netlist)
                   (b-buf-pwr$netlist)))))


(prove-lemma ttl-bidirect-pads-body$unbound-in-body (rewrite)
  (implies
   (lessp l m)
   (and
    (unbound-in-body #i(data l) (ttl-bidirect-pads-body m n pi))
    (unbound-in-body #i(out l) (ttl-bidirect-pads-body m n pi))
    (unbound-in-body #i(po l) (ttl-bidirect-pads-body m n pi))
    (unbound-in-body #i(in l) (ttl-bidirect-pads-body m n pi))))
  ;;Hint
  ((enable unbound-in-body ttl-bidirect-pads-body)))


(disable ttl-bidirect-pads-body$unbound-in-body)
```

```
(prove-lemma ttl-bidirect-pads-body$value (rewrite)
  (implies
   (ttl-bidirect& netlist)
   (and
    (equal (collect-value
            #i(data m n)
            (dual-eval 1 (ttl-bidirect-pads-body m n pi)
                       bindings state-bindings netlist))
           (if (equal (value 'buf-enable bindings) f)
               (v-threefix (collect-value #i(in m n) bindings))
             (if (equal (value 'buf-enable bindings) t)
                 (make-list n (z))
               (make-list n (x)))))

    (equal (collect-value
            #i(out m n)
            (dual-eval 1 (ttl-bidirect-pads-body m n pi)
                       bindings state-bindings netlist))
           (if (equal (value 'buf-enable bindings) f)
               (v-threefix
                (v-wire (collect-value #i(data m n) bindings)
                        (v-threefix (collect-value #i(in m n)
                                                   bindings))))
             (if (equal (value 'buf-enable bindings) t)
                 (v-threefix (collect-value #i(data m n) bindings))
               (make-list n (x)))))))
  ;;Hint
  ((enable ttl-bidirect-pads-body ttl-bidirect-pads-body$unbound-in-body
           v-threefix v-wire ttl-bidirect$value make-list
           f-buf f-not threefix=x ft-buf-rewrite ft-wire-rewrite)
   (disable-theory f-gates)
   (disable threefix)
   (induct
    (ttl-bidirect-pads-body$induction m n pi
                                      bindings state-bindings netlist))))


(disable ttl-bidirect-pads-body$value)
```

```
(prove-lemma ttl-bidirect-pads$value (rewrite)
  (implies
   (and (ttl-bidirect-pads& netlist n)
        (equal (length data) n)
        (properp data)
        (equal (length inputs) n)
        (properp inputs))
   (equal
    (cdr (dual-eval 0 #i(ttl-bidirect-pads n)
                    (list* enable pi (append data inputs))
                    state-bindings netlist))
    (append
     (vft-buf (f-not enable) inputs)
     (v-threefix (v-wire data (vft-buf (f-not enable) inputs))))))
  ;;Hint
  ((enable ttl-bidirect-pads& ttl-bidirect-pads-body$value
           ttl-bidirect-pads*$destructure
           b-buf-pwr$value b-buf$value vft-buf-rewrite)
   (disable v-threefix open-indices append-v-threefix)))


(disable ttl-bidirect-pads$value)
```

14.63 "fm9001-hardware.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;   FM9001-HARDWARE.EVENTS
;;;
;;;   The low-level, 4-valued, functional specification of the FM9001, with
;;;   the test logic disabled.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  Internal states.

;(defn regs           (state)   (nth 0 state))
;(defn flags          (state)   (nth 1 state))


(defn a-reg           (state)   (nth 2 state))

(defn b-reg           (state)   (nth 3 state))

(defn i-reg           (state)   (nth 4 state))

(defn data-out        (state)   (nth 5 state))

(defn addr-out        (state)   (nth 6 state))

(defn reset-          (state)   (nth 7 state))

(defn dtack-          (state)   (nth 8 state))

(defn hold-           (state)   (nth 9 state))

(defn pc-reg          (state)   (nth 10 state))

(defn cntl-state      (state)   (nth 11 state))


(disable a-reg)

(disable b-reg)

(disable i-reg)

(disable data-out)

(disable addr-out)

(disable reset-)

(disable dtack-)
```

```
(disable hold-)

(disable pc-reg)

(disable cntl-state)


(deftheory fm9001-hardware-state-accessors
  (regs flags a-reg b-reg i-reg data-out addr-out reset- dtack- hold-
       pc-reg cntl-state))

;;;  External Inputs.


(defn reset--input   (ext-in)    (nth 0 ext-in))

(defn hold--input    (ext-in)    (nth 1 ext-in))

(defn pc-reg-input   (ext-in)    (subrange ext-in 2 5))


(disable reset--input)

(disable hold--input)

(disable pc-reg-input)


(deftheory fm9001-external-input-accessors
  (reset--input hold--input pc-reg-input))



;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    FM9001-NEXT-STATE
;;;
;;;    The specification of the next state of the processor and memory for a
;;;    single clock cycle.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defn FM9001-next-state (total-state external-inputs)
  (let
    ((p-state (car total-state))
     (mem-state (cadr total-state)))

    (let
      ((regs        (regs       p-state))
       (flags       (flags      p-state))
       (a-reg       (a-reg      p-state))
       (b-reg       (b-reg      p-state))
       (i-reg       (i-reg      p-state))
       (data-out    (data-out   p-state))
       (addr-out    (addr-out   p-state))
       (last-reset- (reset-     p-state))
       (last-dtack- (dtack-     p-state))
       (last-hold-  (hold-      p-state))
       (last-pc-reg (pc-reg     p-state))
       (cntl-state  (cntl-state p-state))

       (reset-    (reset--input external-inputs))
       (hold-     (hold--input  external-inputs))
       (pc-reg-in (pc-reg-input external-inputs)))

      (let
        ((state            (state            (v-threefix cntl-state)))
         (rw-              (rw-              (v-threefix cntl-state)))
         (strobe-          (strobe-          (v-threefix cntl-state)))
         (hdack-           (hdack-           (v-threefix cntl-state)))
         (we-regs          (we-regs          (v-threefix cntl-state)))
         (we-flags         (we-flags         (v-threefix cntl-state)))
         (we-a-reg         (we-a-reg         (v-threefix cntl-state)))
         (we-b-reg         (we-b-reg         (v-threefix cntl-state)))
         (we-i-reg         (we-i-reg         (v-threefix cntl-state)))
         (we-data-out      (we-data-out      (v-threefix cntl-state)))
         (we-addr-out      (we-addr-out      (v-threefix cntl-state)))
         (we-hold-         (we-hold-         (v-threefix cntl-state)))
         (we-pc-reg        (we-pc-reg        (v-threefix cntl-state)))
         (data-in-select   (data-in-select   (v-threefix cntl-state)))
         (dec-addr-out     (dec-addr-out     (v-threefix cntl-state)))
         (select-immediate (select-immediate (v-threefix cntl-state)))
         (regs-address     (regs-address     (v-threefix cntl-state)))
         (alu-c            (alu-c            (v-threefix cntl-state)))
         (alu-op           (alu-op           (v-threefix cntl-state)))
         (alu-zero         (alu-zero         (v-threefix cntl-state)))
         (alu-mpg          (alu-mpg          (v-threefix cntl-state))))

        ;;  ADDR-OUT, RW- and STROBE- are tristate when HOLDing, and pulled
        ;;  high by external pullups.  The EXT- (external) signals are used
        ;;  only by the memory interface.

        (let
          ((ext-addr-out
            (v-pullup (vft-buf (f-buf hdack-)
                               (v-threefix addr-out))))
```

```
 (ext-rw-
  (f-pullup (ft-buf (f-buf hdack-) (f-buf  rw-))))
 (ext-strobe-
  (f-pullup (ft-buf (f-buf hdack-) strobe-)))
 (ext-data-out
  (vft-buf (f-not (f-buf rw-)) (v-threefix data-out))))
(let
  ((mem-response
    (memory-value mem-state ext-strobe- ext-rw-
                  ext-addr-out
                  (make-list 32 (x)))))
  (let
    ((dtack- (car mem-response))
     (ext-data-bus (v-pullup
                    (v-wire ext-data-out (cdr mem-response)))))
    (let
      ((reg-bus (f$extend-immediate
                 select-immediate
                 (a-immediate (v-threefix i-reg))
                 (f$read-regs regs-address regs)))
       (alu-bus (f$core-alu alu-c
                            (v-threefix a-reg)
                            (v-threefix b-reg)
                            alu-zero alu-mpg alu-op
                            (make-tree 32)))
       (data-in (v-threefix (v-wire ext-data-bus ext-data-out))))
      (let
        ((addr-out-bus (f$dec-pass dec-addr-out reg-bus))
         (abi-bus
          (fv-if (f-nand data-in-select
                         (f-not last-dtack-))
                 reg-bus
                 data-in)))
        (list
         (list
          (f$write-regs we-regs regs-address regs (bv alu-bus))
          (f$update-flags flags we-flags alu-bus)
          (fv-if we-a-reg abi-bus a-reg)
          (fv-if we-b-reg abi-bus b-reg)
          (fv-if we-i-reg abi-bus i-reg)
          (fv-if we-data-out (bv alu-bus) data-out)
          (fv-if we-addr-out addr-out-bus addr-out)
          (f-buf reset-)
          (f-or strobe- (f-buf dtack-))
          (f-if we-hold- (f-buf hold-) last-hold-)
          (fv-if we-pc-reg pc-reg-in last-pc-reg)
          (v-threefix (f$next-cntl-state
                       (f-buf last-reset-)
                       (f-buf last-dtack-)
                       (f-buf last-hold-)
                       rw-
                       state
                       (v-threefix i-reg)
                       (v-threefix flags)
                       (v-threefix last-pc-reg)
```

```
                                  regs-address)))
                    (next-memory-state
                     mem-state ext-strobe- ext-rw-
                     ext-addr-out
                     ext-data-bus))))))))))


(disable fm9001-next-state)

;;;  RUN-FM9001 -- Simulates N clock cycles.


(defn run-fm9001 (state inputs n)
  (if (zerop n)
      state
    (run-fm9001 (fm9001-next-state state (car inputs))
                (cdr inputs)
                (sub1 n))))


(disable run-fm9001)


(prove-lemma run-fm9001-base-case (rewrite)
  (implies
   (zerop n)
   (equal (run-fm9001 state inputs n)
          state))
  ;;Hint
  ((enable run-fm9001)))


(prove-lemma run-fm9001-step-case (rewrite)
  (implies
   (not (zerop n))
   (equal (run-fm9001 state inputs n)
          (run-fm9001 (fm9001-next-state state (car inputs))
                      (cdr inputs)
                      (sub1 n))))
  ;;Hint
  ((enable run-fm9001)))


(disable run-fm9001-step-case)
```

```
(prove-lemma run-fm9001-plus (rewrite)
  (equal (run-fm9001 state inputs (plus n m))
         (run-fm9001 (run-fm9001 state inputs n)
                     (nthcdr n inputs)
                     m))
  ;;Hint
  ((enable run-fm9001 nthcdr plus)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    FM9001-STATE-STRUCTURE state
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn fm9001-state-structure (state)
  (and (equal (length state) 2)
       (properp state)
       (equal (length (car state)) 12)
       (properp (car state))
       (equal (length (cadr state)) 8)
       (properp (cadr state))
       (equal (length (caar state)) 4)
       (properp (caar state))))


(prove-lemma fm9001-state-structure$step (rewrite)
  (implies
   (fm9001-state-structure state)
   (fm9001-state-structure (fm9001-next-state state external-inputs)))
  ;;Hint
  ((enable fm9001-next-state)
   (disable-theory f-gates)
   (disable make-list *1*make-list *1*make-tree)))


(prove-lemma fm9001-state-structure$induction (rewrite)
  (implies
   (fm9001-state-structure state)
   (fm9001-state-structure (run-fm9001 state inputs steps)))
  ;;Hint
  ((enable run-fm9001)))
```

```
(prove-lemma fm9001-state-as-a-list ()
  (implies
   (fm9001-state-structure state)
   (equal state
          (list
           (list (list (caaar state)
                       (cadaar state)
                       (caddaar state)
                       (cadddaar state))
                 (cadar state)
                 (caddar state)
                 (cadddar state)
                 (caddddar state)
                 (cadddddar state)
                 (caddddddar state)
                 (cadddddddar state)
                 (caddddddddar state)
                 (cadddddddddar state)
                 (caddddddddddar state)
                 (cadddddddddddar state))
           (list (caadr state)
                 (cadadr state)
                 (caddadr state)
                 (cadddadr state)
                 (caddddadr state)
                 (cadddddadr state)
                 (caddddddadr state)
                 (cadddddddadr state)))))
  ;;Hint
  ((enable equal-length-add1 properp)))
```

14.64 "chip.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    CHIP.EVENTS
;;;
;;;    The DUAL-EVAL model of the FM9001 chip.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    CHIP-MODULE
;;;
;;;    The entire chip, sans pads, as a module.
;;;
;;; INPUTS:
;;;
;;;    CLK  -- The system clock
;;;
;;;    TI   -- Test input (Scan in)
;;;    TE   -- Test enable (Scan enable)
;;;
;;;    DTACK-  --  DTACK input
;;;    RESET-  --  RESET input
;;;    HOLD-   --  HOLD input
;;;
;;;    DISABLE-REGFILE-   --  Must be high to write enable the register
;;;                           file RAM.  During scan-in, DISABLE-REGFILE- is
;;;                           held low to avoid spurious writes to the RAM.
;;;
;;;    TEST-REGFILE-    --  Normally high.  When low, forces the register file
;;;                         output mux to pass data directly from the register
;;;                         file RAM.
;;;
;;;    PC-REG-IN[0..3]    --  The PC-REG inputs.
;;;
;;;    DATA-IN[0..31]  --  The input data bus.  We consider the bus as two
;;;                        unidirectional busses at this level.
;;;
;;; OUTPUTS:
;;;
;;;    TO      --  Test Output (Scan out)
;;;    TIMING  --  The net corresponding to the longest combinational path.
;;;                Used to verify the speed of the device.  This is currently
;;;                the ZERO flag computed by the ALU.
;;;
;;;    HDACK-   --  Hold Acknowledge
;;;
;;;    EN-ADDR-OUT-  -- The inverse of HDACK-, with high drive for ADDR-BUS
;;;                     and memory control lines.
;;;
```

```
;;;    RW-      --  Memory control lines  (High Drive for DATA-OUT)
;;;    STROBE-
;;;
;;;    ADDR-OUT[0..31]  --  Address bus
;;;    DATA-OUT[0..31]  --  Data output bus
;;;
;;;    FLAGS[0..3]     --  Observability pins
;;;    CNTL-STATE[0..4]
;;;    I-REG[28..31]
;;;
;;;
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(module-generator
 (chip-module*)

 'chip-module

 (list* 'clk 'ti 'te 'dtack- 'reset- 'hold-
        'disable-regfile- 'test-regfile-
        (append #i(pc-reg-in 0 4) #i(data-in 0 32)))

 (list* 'to 'timing 'hdack- 'en-addr-out- 'rw- 'strobe-
        (append #i(addr-out 0 32)
                (append #i(data-out 0 32)
                        (append #i(flags 0 4)
                                (append #i(cntl-state 0 5)
                                        (subrange #i(i-reg 0 32) 28 31))))))
 (list

  ;;  The sequential modules, in specification order except for CNTL-STATE,
  ;;  which must appear first since its combinational outputs control the
  ;;  combinational output of the register file.

  ;;  CNTL-STATE.  Scan TI to #i(ALU-MPG 6)
  (list 'cntl-state
        (list* 'rw-sig- 'strobe- 'hdack-
               'we-regs 'we-a-reg 'we-b-reg 'we-i-reg
               'we-data-out 'we-addr-out 'we-hold- 'we-pc-reg
               'data-in-select 'dec-addr-out 'select-immediate
               'alu-c 'alu-zero
               (append #i(cntl-state 0 5)
                       (append #i(we-flags 0 4)
                               (append #i(regs-address 0 4)
                                       (append #i(alu-op 0 4)
                                               #i(alu-mpg 0 7))))))
        #i(reg 40)
        (list* 'clk 'te-sig 'ti #i(next-state 0 40)))

  ;;  REGS.  Scan #i(ALU-MPG 6) to REGFILE-TO
  (list 'regs
        (cons 'regfile-to #i(regfile-out 0 32))
        'regfile
        (list* 'clk 'te-sig #i(alu-mpg 6)
               'we-regs 'disable-regfile- 'test-regfile-
               (append #i(regs-address 0 4) (bv #i(alu-bus 0 35)))))

  ;;  FLAGS.  Scan REGFILE-TO to #i(FLAGS 3)
  (list 'cvnz-flags
        #i(flags 0 4)
        'flags
        (list* 'clk 'te-sig 'regfile-to (append #i(we-flags 0 4)
                                               #i(alu-bus 0 35))))
  ;;  A-REG.  Scan #i(FLAGS 3) to #i(A-REG 31)
  (list 'a-reg
        #i(a-reg 0 32)
        #i(we-reg 32)
```

```
      (list* 'clk 'we-a-reg 'te-sig #i(FLAGS 3) #i(abi-bus 0 32)))
;;  B-REG.  Scan #i(A-REG 31) to #i(B-REG 31)
(list 'b-reg
      #i(b-reg 0 32)
      #i(we-reg 32)
      (list* 'clk 'we-b-reg 'te-sig #i(a-reg 31) #i(abi-bus 0 32)))
;;  I-REG.  Scan #i(B-REG 31) to #i(I-REG 31)
(list 'i-reg
      #i(i-reg 0 32)
      #i(we-reg 32)
      (list* 'clk 'we-i-reg 'te-sig #i(b-reg 31) #i(abi-bus 0 32)))
;;  DATA-OUT.  Scan #i(I-REG 31) to #i(DATA-OUT 31)
(list 'data-out
      #i(data-out 0 32)
      #i(we-reg 32)
      (list* 'clk 'we-data-out 'te-sig #i(i-reg 31) (bv #i(alu-bus 0 35))))
;;  ADDR-OUT.  Scan #i(DATA-OUT 31) to #i(ADDR-OUT 31)
(list 'addr-out
      #i(addr-out 0 32)
      #i(we-reg 32)
      (list* 'clk 'we-addr-out 'te-sig #i(data-out 31)
             #i(addr-out-bus 0 32)))
;;  RESET.  Scan #i(ADDR-OUT 31) to LAST-RESET-
(list 'reset-latch
      '(last-reset- last-reset-inv)
      'fd1s
      (list 'reset- 'clk #i(addr-out 31) 'te-sig))
;;  DTACK.  Scan LAST-RESET- to LAST-DTACK-
;;
;;  The DTACK- input is ORed with STROBE-, to insure that the DTACK- state
;;  is initializable, and remains boolean.
'(dtack--or (dtack--or-strobe-) b-or (strobe- dtack-))
(list 'dtack-latch
      '(last-dtack- last-dtack-inv)
      'fd1s
      '(dtack--or-strobe- clk last-reset- te-sig))
;;  HOLD.  Scan LAST-DTACK- to LAST-HOLD-
(list 'hold-latch
      '(last-hold- last-hold-inv)
      'fd1slp
      '(hold- clk we-hold- last-dtack- te-sig))

;;  PC-REG.  Scan LAST-HOLD- to #i(PC-REG 3) [Renamed to TO later].
(list 'pc-reg
      #i(pc-reg 0 4)
      #i(we-reg 4)
      (list* 'clk 'we-pc-reg 'te-sig 'last-hold- #i(pc-reg-in 0 4)))

;;  Now, the combinational modules.

;;  IMMEDIATE/PASS
(list 'immediate-pass
      #i(reg-bus 0 32)
      'extend-immediate
      (cons 'select-immediate
```

```
            (append (a-immediate #i(i-reg 0 32))
                    #i(regfile-out 0 32))))
;;  DEC/PASS
(list 'dec-pass
      #i(addr-out-bus 0 32)
      #i(dec-pass 32)
      (cons 'dec-addr-out #i(reg-bus 0 32)))
;;  DATA-IN-MUX
;;
;;  The data input bus is only selected when DTACK- is asserted, in order to
;;  avoid passing garbage onto the ABI-BUS.
'(mux-cntl (abi-cntl) b-nand (data-in-select last-dtack-inv))
(list 'data-in-mux
      #i(abi-bus 0 32)
      #i(tv-if (tree-number (make-tree 32)))
      (cons 'abi-cntl (append #i(reg-bus 0 32) #i(data-in 0 32))))
;;  ALU
(list 'alu
      #i(alu-bus 0 35)
      #i(core-alu (tree-number (make-tree 32)))
      (cons 'alu-c (append #i(a-reg 0 32)
                           (append #i(b-reg 0 32)
                                   (cons 'alu-zero
                                         (append #i(alu-mpg 0 7)
                                                 #i(alu-op 0 4)))))))

;;  NEXT-STATE
(list 'next-state
      #i(next-state 0 40)
      'next-cntl-state
      (list* 'last-reset- 'last-dtack- 'last-hold- 'rw-sig-
             (append #i(cntl-state 0 5)
                     (append #i(i-reg 0 32)
                             (append #i(flags 0 4)
                                     (append #i(pc-reg 0 4)
                                             #i(regs-address 0 4)))))))

;;  TE is buffered due to its high load.

'(te-buffer (te-sig) b-buf-pwr (te))

;;  RW- is buffered to enable the DATA-OUT pads.

'(rw-buffer (rw-) b-buf (rw-sig-))

;;  ADDR-OUT, RW- and STROBE- drivers are enabled whenever HDACK- is not
;;  asserted.

'(en-addr-out-gate (en-addr-out-) b-not (hdack-))

;;  We believe the longest combinational net is the ALU ZERO
;;  computation.

(list 'timing-gate '(timing) 'id (list #i(alu-bus 2)))
```

```
  ;;  Rename the scan out

  (list 'scanout '(to) 'id (list #i(pc-reg 3))))

 ;;  States

 '(regs cvnz-flags a-reg b-reg i-reg data-out addr-out
        reset-latch dtack-latch hold-latch
        pc-reg cntl-state))



(disable *1*chip-module*)


(defn chip-module& (netlist)
  (and (equal (lookup-module 'chip-module netlist)
              (chip-module*))
       (let ((netlist (delete-module 'chip-module netlist)))
         (and
          (regfile& netlist)
          (flags& netlist)
          (we-reg& netlist 32)
          (fd1s& netlist)
          (fd1slp& netlist)
          (we-reg& netlist 4)
          (reg& netlist 40)
          (id& netlist)
          (b-buf-pwr& netlist)
          (b-buf& netlist)
          (b-not& netlist)
          (extend-immediate& netlist)
          (dec-pass& netlist 32)
          (tv-if& netlist (make-tree 32))
          (core-alu& netlist (make-tree 32))
          (next-cntl-state& netlist)))))


(disable chip-module&)
```

```
(defn chip-module$netlist ()
  (cons
   (chip-module*)
   (union (regfile$netlist)
          (union (union (union (union (flags$netlist)
                                      (we-reg$netlist 32))
                               (union (fd1s$netlist)
                                      (fd1slp$netlist)))
                        (union (union (union (we-reg$netlist 4)
                                             (reg$netlist 40))
                                      (union (id$netlist)
                                             (extend-immediate$netlist)))
                               (union (union (dec-pass$netlist 32)
                                             (tv-if$netlist (make-tree 32)))
                                      (union (core-alu$netlist (make-tree 32))
                                             (next-cntl-state$netlist)))))
                 (union (b-buf-pwr$netlist)
                        (union (b-buf$netlist)
                               (b-not$netlist)))))))

;;;  A few crocks for the coming proof.


(prove-lemma equal-length-40-as-collected-nth+subrange ()
  (implies
   (and (equal (length l) 40)
        (properp l))
   (equal l (append (list-as-collected-nth l 16 0)
                    (subrange l 16 39))))
  ;;Hint
  ((enable open-nth properp-as-null-nthcdr our-car-cdr-elim
           open-subrange)
   (disable car-cdr-elim)))
```

```
(prove-lemma list-as-cntl-state-crock ()
  (implies
   (and (properp list)
        (equal (length list) 40))
   (equal list
          (list*
           (rw-             list)
           (strobe-         list)
           (hdack-          list)
           (we-regs         list)
           (we-a-reg        list)
           (we-b-reg        list)
           (we-i-reg        list)
           (we-data-out     list)
           (we-addr-out     list)
           (we-hold-        list)
           (we-pc-reg       list)
           (data-in-select  list)
           (dec-addr-out    list)
           (select-immediate list)
           (alu-c           list)
           (alu-zero        list)
           (append (state list)
                   (append (we-flags list)
                           (append (regs-address list)
                                   (append (alu-op list)
                                           (alu-mpg list)))))))))
  ;;Hint
  ((enable-theory control-state-accessor-theory)
   (enable open-subrange)
   (use (equal-length-40-as-collected-nth+subrange (l list)))))
```

```
(prove-lemma reg-40$value-as-cntl-state (rewrite)
  (implies
   (and (reg& netlist 40)
        (properp state)
        (equal (length state) 40))
   (equal (dual-eval 0 #i(reg 40) args state netlist)
          (list*
           (rw-              (v-threefix state))
           (strobe-          (v-threefix state))
           (hdack-           (v-threefix state))
           (we-regs          (v-threefix state))
           (we-a-reg         (v-threefix state))
           (we-b-reg         (v-threefix state))
           (we-i-reg         (v-threefix state))
           (we-data-out      (v-threefix state))
           (we-addr-out      (v-threefix state))
           (we-hold-         (v-threefix state))
           (we-pc-reg        (v-threefix state))
           (data-in-select   (v-threefix state))
           (dec-addr-out     (v-threefix state))
           (select-immediate (v-threefix state))
           (alu-c            (v-threefix state))
           (alu-zero         (v-threefix state))
           (append (state (v-threefix state))
                   (append (we-flags (v-threefix state))
                           (append (regs-address (v-threefix state))
                                   (append (alu-op (v-threefix state))
                                           (alu-mpg (v-threefix state))))))))))
  ;;Hint
  ((enable reg$value)
   (use (list-as-cntl-state-crock (list (v-threefix state))))))


(disable reg-40$value-as-cntl-state)


(prove-lemma bv-as-subrange (rewrite)
  (implies
   (and (geq (length x) 3)
        (properp x))
   (equal (bv x) (subrange x 3 (sub1 (length x)))))
  ;;Hint
  ((enable bv subrange subrange-0)))


(disable bv-as-subrange)

;;;  Lemmas
```

```
(defn machine-state-invariant (machine-state)
  (let
    ((regs              (car machine-state))
     (flags             (cadr machine-state))
     (a-reg             (caddr machine-state))
     (b-reg             (cadddr machine-state))
     (i-reg             (caddddr machine-state))
     (data-out          (cadddddr machine-state))
     (addr-out          (caddddddr machine-state))
     (last-reset-       (cadddddddr machine-state))
     (last-dtack-       (caddddddddr machine-state))
     (last-hold-        (cadddddddddr machine-state))
     (pc-reg            (caddddddddddr machine-state))
     (cntl-state        (cadddddddddddr machine-state)))
    (let
      ((regs-regs    (car regs))
       (regs-we      (cadr regs))
       (regs-data    (caddr regs))
       (regs-address (cadddr regs)))
      (and
       (all-ramp-mem 4 regs-regs)
       (memory-properp 4 32 regs-regs)
       (properp regs-data) (equal (length regs-data) 32)
       (properp regs-address) (equal (length regs-address) 4)
       (properp flags) (equal (length flags) 4)
       (properp a-reg) (equal (length a-reg) 32)
       (properp b-reg) (equal (length b-reg) 32)
       (properp i-reg) (equal (length i-reg) 32)
       (properp data-out) (equal (length data-out) 32)
       (properp addr-out) (equal (length addr-out) 32)
       (properp pc-reg) (equal (length pc-reg) 4)
       (properp cntl-state) (equal (length cntl-state) 40)))))
```

```
(prove-lemma chip-module$state (rewrite)
  (let
    ((regs (list regs-regs regs-we regs-data regs-addr)))
    (let
      ((machine-state
        (list
         regs flags a-reg b-reg i-reg data-out addr-out
         last-reset- last-dtack- last-hold- last-pc-reg
         cntl-state))
       (inputs
        (list* clk ti te dtack- reset- hold-
               disable-regfile- test-regfile-
               (append pc-reg-in data-in))))
      (implies
       (and (chip-module& netlist)
            (machine-state-invariant machine-state)
            (equal te f)
            (equal disable-regfile- t)
            (equal test-regfile- t)
            (properp pc-reg-in) (equal (length pc-reg-in) 4)
            (properp data-in) (equal (length data-in) 32))
       (equal (dual-eval 2 'chip-module inputs machine-state netlist)
              (let
                ((state            (state            (v-threefix cntl-state)))
                 (rw-              (rw-              (v-threefix cntl-state)))
                 (strobe-          (strobe-          (v-threefix cntl-state)))
                 (hdack-           (hdack-           (v-threefix cntl-state)))
                 (we-regs          (we-regs          (v-threefix cntl-state)))
                 (we-flags         (we-flags         (v-threefix cntl-state)))
                 (we-a-reg         (we-a-reg         (v-threefix cntl-state)))
                 (we-b-reg         (we-b-reg         (v-threefix cntl-state)))
                 (we-i-reg         (we-i-reg         (v-threefix cntl-state)))
                 (we-data-out      (we-data-out      (v-threefix cntl-state)))
                 (we-addr-out      (we-addr-out      (v-threefix cntl-state)))
                 (we-hold-         (we-hold-         (v-threefix cntl-state)))
                 (we-pc-reg        (we-pc-reg        (v-threefix cntl-state)))
                 (data-in-select   (data-in-select   (v-threefix cntl-state)))
                 (dec-addr-out     (dec-addr-out     (v-threefix cntl-state)))
                 (select-immediate (select-immediate (v-threefix cntl-state)))
                 (regs-address     (regs-address     (v-threefix cntl-state)))
                 (alu-c            (alu-c            (v-threefix cntl-state)))
                 (alu-op           (alu-op           (v-threefix cntl-state)))
                 (alu-zero         (alu-zero         (v-threefix cntl-state)))
                 (alu-mpg          (alu-mpg          (v-threefix cntl-state))))
                (let
                  ((reg-bus (f$extend-immediate
                             select-immediate
                             (a-immediate (v-threefix i-reg))
                             (f$read-regs regs-address regs)))
                   (alu-bus (f$core-alu alu-c
                                        (v-threefix a-reg)
                                        (v-threefix b-reg)
                                        alu-zero alu-mpg alu-op
                                        (make-tree 32))))
```

```
                (let
                  ((addr-out-bus (f$dec-pass dec-addr-out reg-bus))
                   (abi-bus (fv-if (f-nand data-in-select
                                           (f-not last-dtack-))
                                   reg-bus
                                   data-in)))
                  (list
                   (f$write-regs we-regs regs-address regs (bv alu-bus))
                   (f$update-flags flags we-flags alu-bus)
                   (fv-if we-a-reg abi-bus a-reg)
                   (fv-if we-b-reg abi-bus b-reg)
                   (fv-if we-i-reg abi-bus i-reg)
                   (fv-if we-data-out (bv alu-bus) data-out)
                   (fv-if we-addr-out addr-out-bus addr-out)
                   (f-buf reset-)
                   (f-or strobe- dtack-)
                   (f-if we-hold- hold- last-hold-)
                   (fv-if we-pc-reg pc-reg-in last-pc-reg)
                   (v-threefix (f$next-cntl-state
                                (f-buf last-reset-)
                                (f-buf last-dtack-)
                                (f-buf last-hold-)
                                rw-
                                state
                                (v-threefix i-reg)
                                (v-threefix flags)
                                (v-threefix last-pc-reg)
                                regs-address))))))))))
;;Hint
((enable chip-module& chip-module*$destructure
         regfile$value regfile$state flags$value flags$state
         we-reg$value we-reg$state fd1s$value fd1s$state
         fd1slp$value fd1slp$state reg$value reg$state id$value
         b-buf-pwr$value b-buf$value b-not$value dec-pass$value tv-if$value
         core-alu$value next-cntl-state$value extend-immediate$value
         reg-40$value-as-cntl-state b-or$value b-nand$value
         bv-as-subrange a-immediate f-buf-delete-lemmas)
 (disable indices *1*indices open-indices nth open-nth *1*nth
          make-tree *1*make-tree threefix f-gates=b-gates
          open-v-threefix)
 (disable-theory f-gates b-gates)
 (enable-theory control-state-accessor-theory)))
```

```
(prove-lemma chip-module$value (rewrite)
  (let
    ((regs (list regs-regs regs-we regs-data regs-addr)))
    (let
      ((machine-state
        (list
         regs flags a-reg b-reg i-reg data-out addr-out
         last-reset- last-dtack- last-hold- last-pc-reg
         cntl-state))
       (inputs
        (list* clk ti te dtack- reset- hold-
               disable-regfile- test-regfile-
               (append pc-reg-in data-in))))
      (implies
       (and (chip-module& netlist)
            (machine-state-invariant machine-state)
            (properp pc-reg-in) (equal (length pc-reg-in) 4)
            (properp data-in) (equal (length data-in) 32))
       (equal (dual-eval 0 'chip-module inputs machine-state netlist)
              (let
                ((state           (state           (v-threefix cntl-state)))
                 (rw-             (rw-             (v-threefix cntl-state)))
                 (strobe-         (strobe-         (v-threefix cntl-state)))
                 (hdack-          (hdack-          (v-threefix cntl-state)))
                 (we-regs         (we-regs         (v-threefix cntl-state)))
                 (we-flags        (we-flags        (v-threefix cntl-state)))
                 (we-a-reg        (we-a-reg        (v-threefix cntl-state)))
                 (we-b-reg        (we-b-reg        (v-threefix cntl-state)))
                 (we-i-reg        (we-i-reg        (v-threefix cntl-state)))
                 (we-data-out     (we-data-out     (v-threefix cntl-state)))
                 (we-addr-out     (we-addr-out     (v-threefix cntl-state)))
                 (we-hold-        (we-hold-        (v-threefix cntl-state)))
                 (we-pc-reg       (we-pc-reg       (v-threefix cntl-state)))
                 (data-in-select  (data-in-select  (v-threefix cntl-state)))
                 (dec-addr-out    (dec-addr-out    (v-threefix cntl-state)))
                 (select-immediate (select-immediate (v-threefix cntl-state)))
                 (regs-address    (regs-address    (v-threefix cntl-state)))
                 (alu-c           (alu-c           (v-threefix cntl-state)))
                 (alu-op          (alu-op          (v-threefix cntl-state)))
                 (alu-zero        (alu-zero        (v-threefix cntl-state)))
                 (alu-mpg         (alu-mpg         (v-threefix cntl-state))))
                (let
                  ((reg-bus (f$extend-immediate
                             select-immediate
                             (a-immediate (v-threefix i-reg))
                             (f$read-regs regs-address regs)))
                   (alu-bus (f$core-alu alu-c
                                        (v-threefix a-reg)
                                        (v-threefix b-reg)
                                        alu-zero alu-mpg alu-op
                                        (make-tree 32))))
                  (let
                    ((addr-out-bus (f$dec-pass dec-addr-out reg-bus))
                     (abi-bus (fv-if (f-nand data-in-select
```

```
                                        (f-not last-dtack-))
                                   reg-bus
                                   data-in)))
                    (list* (nth 3 (v-threefix last-pc-reg))
                           (nth 2 alu-bus)
                           hdack-
                           (f-not hdack-)
                           (f-buf rw-)
                           strobe-
                           (append5
                            (v-threefix addr-out)
                            (v-threefix data-out)
                            (v-threefix flags)
                            state
                            (subrange (v-threefix i-reg) 28 31))))))))))
  ;;Hint
  ((enable chip-module& chip-module*$destructure
          regfile$value regfile$state flags$value flags$state
          we-reg$value we-reg$state fd1s$value fd1s$state
          fd1slp$value fd1slp$state reg$value reg$state id$value
          b-buf-pwr$value b-buf$value b-not$value dec-pass$value tv-if$value
          core-alu$value next-cntl-state$value extend-immediate$value
          reg-40$value-as-cntl-state b-or$value b-nand$value
          bv-as-subrange a-immediate f-buf-delete-lemmas)
   (disable indices *1*indices open-indices nth open-nth *1*nth
           make-tree *1*make-tree threefix f-gates=b-gates
           open-v-threefix)
   (disable-theory f-gates b-gates)
   (enable-theory control-state-accessor-theory)))



;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    CHIP
;;;
;;;    The entire chip, with pads, as a module.
;;;
;;; INPUTS:
;;;
;;;  The system clock input is a special two-pad, clock-driver buffer.
;;;  The other inputs are ordinary TTL inputs.
;;;
;;;    CLK  -- The system clock
;;;
;;;    TI   -- Test input (Scan in)
;;;    TE   -- Test enable (Scan enable)
;;;
;;;    DTACK-  --  DTACK input
;;;    RESET-  --  RESET input
;;;    HOLD-   --  HOLD input
;;;
;;;    DISABLE-REGFILE-   --  Must be high to write enable the register
;;;                           file RAM.  During scan-in, DISABLE-REGFILE- is
;;;                           held low to avoid spurious writes to the RAM.
```

```
;;;
;;;    TEST-REGFILE-    --  Normally high.  When low, forces the register file
;;;                         output mux to pass data directly from the RAM.
;;;
;;;    PC-REG-IN[0..3]  --  The PC-REG inputs.
;;;
;;;
;;; BI-DIRECTS:
;;;
;;;    DATA-BUS[0..31]  --  The bi-directional data bus. (Controlled by RW-)
;;;
;;; OUTPUTS:
;;;
;;;  The memory control lines and the ADDR-OUT lines are tri-state.
;;;  The other output signals are ordinary TTL outputs.
;;;
;;;    PO      --  Parametric Output.
;;;    TO      --  Test Output (Scan out)
;;;    TIMING  --  The net corresponding to the longest combinational path.
;;;                Used to verify the speed of the device.
;;;
;;;    HDACK-   --  Hold Acknowledge
;;;
;;;    RW-      --  Memory control lines  (Tristate)
;;;    STROBE-
;;;
;;;    ADDR-OUT[0..31]  --  Address bus (Tristate)
;;;
;;;    FLAGS[0..3]     --  Observability pins
;;;    CNTL-STATE[0..4]
;;;    I-REG[28..31]
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(module-generator
 (chip*)

 'chip

 (list* 'clk 'ti 'te 'dtack- 'reset- 'hold-
        'disable-regfile- 'test-regfile-
        (append #i(pc-reg-in 0 4)
                #i(data-bus 0 32)))

 (list* 'po 'to 'timing 'hdack- 'rw- 'strobe-
        (append #i(addr-out 0 32)
                (append #i(data-bus 0 32)
                        (append #i(flags 0 4)
                                (append #i(cntl-state 0 5)
                                        #i(i-reg 28 4))))))
 (list

  ;;  The processor core.

  (list 'body
        (list* 'i-to 'i-timing 'i-hdack- 'i-en-addr-out- 'i-rw- 'i-strobe-
               (append #i(i-addr-out 0 32)
                       (append #i(i-data-out 0 32)
                               (append #i(i-flags 0 4)
                                       (append #i(i-cntl-state 0 5)
                                               #i(i-i-reg 28 4))))))

        'chip-module
        (list* 'i-clk 'i-ti 'i-te 'i-dtack- 'i-reset-
               'i-hold- 'i-disable-regfile- 'i-test-regfile-
               (append #i(i-pc-reg 0 4)
                       #i(i-data-in 0 32))))

  ;;  The input pads.  The pads are ordered such that the inputs to PROCMON are
  ;;  not from bidirect pads (as required), while maintaining the ordering
  ;;  required by the DUAL-EVAL interpreter.

  '(plus-5 (b-true-p) vdd-parametric())

  '(clock-pad (i-clk clk-po) ttl-clk-input (clk b-true-p))
  '(ti-pad    (i-ti  ti-po)  ttl-input     (ti b-true-p))
  '(te-pad    (i-te  te-po)  ttl-input     (te ti-po))

  '(dtack-pad (i-dtack- dtack-po) ttl-input (dtack- te-po))
  '(reset-pad (i-reset- reset-po) ttl-input (reset- dtack-po))
  '(hold-pad  (i-hold- hold-po)   ttl-input (hold- reset-po))

  '(disable-regfile-pad (i-disable-regfile- disable-regfile-po)
                        ttl-input (disable-regfile- hold-po))
  '(test-regfile-pad (i-test-regfile- test-regfile-po)
                     ttl-input (test-regfile- disable-regfile-po))

  ;;  DATA-BUS
```

```
(list 'data-bus-pads
      (cons 'data-bus-po (append #i(data-bus 0 32) #i(i-data-in 0 32)))
      #i(ttl-bidirect-pads 32)
      (list* 'i-rw- 'test-regfile-po
             (append #i(data-bus 0 32) #i(i-data-out 0 32))))

;;  PC-REG

(list 'pc-reg-pads
      (cons 'pc-reg-po #i(i-pc-reg 0 4))
      #i(ttl-input-pads 4)
      (cons 'data-bus-po #i(pc-reg-in 0 4)))

;;  The process monitor -- required by LSI Logic.

'(monitor (i-po) procmon (pc-reg-po clk-po b-true-p b-true-p))

;;  The output pads.

'(po-pad     (po)      ttl-output-parametric (i-po))
'(to-pad     (to)      ttl-output (i-to))
'(timing-pad (timing)  ttl-output-fast (i-timing))

'(hdack-pad   (hdack-)  ttl-output-fast (i-hdack-))
'(rw-pad      (rw-)     ttl-tri-output-fast (i-rw- i-en-addr-out-))
'(strobe-pad  (strobe-) ttl-tri-output-fast (i-strobe- i-en-addr-out-))

;;  ADDR-OUT

(list 'addr-out-pads
      #i(addr-out 0 32)
      #i(ttl-tri-output-pads 32)
      (cons 'i-en-addr-out- #i(i-addr-out 0 32)))

;;  FLAGS

(list 'flags-pads
      #i(flags 0 4)
      #i(ttl-output-pads 4)
      #i(i-flags 0 4))

;;  CNTL-STATE

(list 'cntl-state-pads
      #i(cntl-state 0 5)
      #i(ttl-output-pads 5)
      #i(i-cntl-state 0 5))

;;  I-REG [28..31]

(list 'i-reg-pads
      #i(i-reg 28 4)
      #i(ttl-output-pads 4)
      #i(i-i-reg 28 4))
```

```
  )
 ;;  The processor state

 'body)


(disable *1*chip*)




(defn chip& (netlist)
  (and (equal (lookup-module 'chip netlist)
              (chip*))
       (let ((netlist (delete-module 'chip netlist)))
         (and
          (vdd-parametric& netlist)
          (ttl-clk-input& netlist)
          (ttl-input& netlist)
          (ttl-input-pads& netlist 4)
          (chip-module& netlist)
          (ttl-bidirect-pads& netlist 32)
          (ttl-output-parametric& netlist)
          (ttl-output& netlist)
          (ttl-output-fast& netlist)
          (ttl-tri-output-fast& netlist)
          (ttl-tri-output-pads& netlist 32)
          (ttl-output-pads& netlist 4)
          (ttl-output-pads& netlist 5)
          (procmon& netlist)))))


(disable chip&)


(defn chip$netlist ()
  (cons
   (chip*)
   (union (union (union (union (ttl-bidirect-pads$netlist 32)
                               (chip-module$netlist))
                        (union (ttl-input$netlist)
                               (ttl-input-pads$netlist 4)))
                 (union (union (ttl-clk-input$netlist)
                               (vdd-parametric$netlist))
                        (union (ttl-output-parametric$netlist)
                               (ttl-output$netlist))))
          (union (union (union (ttl-output-fast$netlist)
                               (ttl-tri-output-fast$netlist))
                        (union (ttl-tri-output-pads$netlist 32)
                               (ttl-output-pads$netlist 4)))
                 (union (ttl-output-pads$netlist 5)
                        (procmon$netlist))))))
```

```
(prove-lemma chip$state (rewrite)
  (let
    ((regs (list regs-regs regs-we regs-data regs-addr)))
    (let
      ((machine-state
        (list
         regs flags a-reg b-reg i-reg data-out addr-out
         last-reset- last-dtack- last-hold- last-pc-reg
         cntl-state))
       (inputs
        (list* clk ti te dtack- reset- hold-
               disable-regfile- test-regfile-
               (append pc-reg-in data-in))))
      (implies
       (and (chip& netlist)
            (machine-state-invariant machine-state)
            (equal te f)
            (equal disable-regfile- t)
            (equal test-regfile- t)
            (properp pc-reg-in) (equal (length pc-reg-in) 4)
            (properp data-in) (equal (length data-in) 32))
       (equal (dual-eval 2 'chip inputs machine-state netlist)
              (let
                ((state            (state            (v-threefix cntl-state)))
                 (rw-              (rw-              (v-threefix cntl-state)))
                 (strobe-          (strobe-          (v-threefix cntl-state)))
                 (hdack-           (hdack-           (v-threefix cntl-state)))
                 (we-regs          (we-regs          (v-threefix cntl-state)))
                 (we-flags         (we-flags         (v-threefix cntl-state)))
                 (we-a-reg         (we-a-reg         (v-threefix cntl-state)))
                 (we-b-reg         (we-b-reg         (v-threefix cntl-state)))
                 (we-i-reg         (we-i-reg         (v-threefix cntl-state)))
                 (we-data-out      (we-data-out      (v-threefix cntl-state)))
                 (we-addr-out      (we-addr-out      (v-threefix cntl-state)))
                 (we-hold-         (we-hold-         (v-threefix cntl-state)))
                 (we-pc-reg        (we-pc-reg        (v-threefix cntl-state)))
                 (data-in-select   (data-in-select   (v-threefix cntl-state)))
                 (dec-addr-out     (dec-addr-out     (v-threefix cntl-state)))
                 (select-immediate (select-immediate (v-threefix cntl-state)))
                 (regs-address     (regs-address     (v-threefix cntl-state)))
                 (alu-c            (alu-c            (v-threefix cntl-state)))
                 (alu-op           (alu-op           (v-threefix cntl-state)))
                 (alu-zero         (alu-zero         (v-threefix cntl-state)))
                 (alu-mpg          (alu-mpg          (v-threefix cntl-state))))
                (let
                  ((reg-bus (f$extend-immediate
                             select-immediate
                             (a-immediate (v-threefix i-reg))
                             (f$read-regs regs-address regs)))
                   (alu-bus (f$core-alu alu-c
                                        (v-threefix a-reg)
                                        (v-threefix b-reg)
                                        alu-zero alu-mpg alu-op
                                        (make-tree 32))))
```

```
                 (let
                   ((addr-out-bus (f$dec-pass dec-addr-out reg-bus))
                    (abi-bus
                     (fv-if (f-nand data-in-select
                                    (f-not last-dtack-))
                            reg-bus
                            (v-wire data-in
                                    (vft-buf (f-not (f-buf rw-))
                                             (v-threefix data-out))))))
                   (list
                    (f$write-regs we-regs regs-address regs (bv alu-bus))
                    (f$update-flags flags we-flags alu-bus)
                    (fv-if we-a-reg abi-bus a-reg)
                    (fv-if we-b-reg abi-bus b-reg)
                    (fv-if we-i-reg abi-bus i-reg)
                    (fv-if we-data-out (bv alu-bus) data-out)
                    (fv-if we-addr-out addr-out-bus addr-out)
                    (f-buf reset-)
                    (f-or strobe- (f-buf dtack-))
                    (f-if we-hold- (f-buf hold-) last-hold-)
                    (fv-if we-pc-reg pc-reg-in last-pc-reg)
                    (v-threefix (f$next-cntl-state
                                 (f-buf last-reset-)
                                 (f-buf last-dtack-)
                                 (f-buf last-hold-)
                                 rw-
                                 state
                                 (v-threefix i-reg)
                                 (v-threefix flags)
                                 (v-threefix last-pc-reg)
                                 regs-address))))))))))
 ;;Hint
 ((enable chip& chip*$destructure
          chip-module$value chip-module$state
          vdd-parametric$value
          ttl-clk-input$value ttl-input$value
          ttl-output$value ttl-tri-output$value
          ttl-output-fast$value ttl-tri-output-fast$value
          ttl-input-pads$value ttl-output-pads$value
          ttl-bidirect-pads$value ttl-tri-output-pads$value
          f-buf-delete-lemmas)
  (disable indices *1*indices open-indices nth open-nth *1*nth
           make-tree *1*make-tree threefix f-gates=b-gates
           open-v-threefix append-v-threefix)
  (disable-theory f-gates b-gates)))
```

```
(prove-lemma chip$value (rewrite)
  (let
    ((regs (list regs-regs regs-we regs-data regs-addr)))
    (let
      ((machine-state
        (list
         regs flags a-reg b-reg i-reg data-out addr-out
         last-reset- last-dtack- last-hold- last-pc-reg
         cntl-state))
       (inputs
        (list* clk ti te dtack- reset- hold-
               disable-regfile- test-regfile-
               (append pc-reg-in data-in))))
      (implies
       (and (chip& netlist)
            (machine-state-invariant machine-state)
            (properp pc-reg-in) (equal (length pc-reg-in) 4)
            (properp data-in) (equal (length data-in) 32))
       (equal (cdr (dual-eval 0 'chip inputs machine-state netlist))
              (let
                ((state            (state            (v-threefix cntl-state)))
                 (rw-              (rw-              (v-threefix cntl-state)))
                 (strobe-          (strobe-          (v-threefix cntl-state)))
                 (hdack-           (hdack-           (v-threefix cntl-state)))
                 (we-regs          (we-regs          (v-threefix cntl-state)))
                 (we-flags         (we-flags         (v-threefix cntl-state)))
                 (we-a-reg         (we-a-reg         (v-threefix cntl-state)))
                 (we-b-reg         (we-b-reg         (v-threefix cntl-state)))
                 (we-i-reg         (we-i-reg         (v-threefix cntl-state)))
                 (we-data-out      (we-data-out      (v-threefix cntl-state)))
                 (we-addr-out      (we-addr-out      (v-threefix cntl-state)))
                 (we-hold-         (we-hold-         (v-threefix cntl-state)))
                 (we-pc-reg        (we-pc-reg        (v-threefix cntl-state)))
                 (data-in-select   (data-in-select   (v-threefix cntl-state)))
                 (dec-addr-out     (dec-addr-out     (v-threefix cntl-state)))
                 (select-immediate (select-immediate (v-threefix cntl-state)))
                 (regs-address     (regs-address     (v-threefix cntl-state)))
                 (alu-c            (alu-c            (v-threefix cntl-state)))
                 (alu-op           (alu-op           (v-threefix cntl-state)))
                 (alu-zero         (alu-zero         (v-threefix cntl-state)))
                 (alu-mpg          (alu-mpg          (v-threefix cntl-state))))
                (let
                  ((reg-bus (f$extend-immediate
                             select-immediate
                             (a-immediate (v-threefix i-reg))
                             (f$read-regs regs-address regs)))
                   (alu-bus (f$core-alu alu-c
                                        (v-threefix a-reg)
                                        (v-threefix b-reg)
                                        alu-zero alu-mpg alu-op
                                        (make-tree 32))))
                  (let
                    ((addr-out-bus (f$dec-pass dec-addr-out reg-bus))
                     (abi-bus
```

```
                    (fv-if (f-nand data-in-select
                                   (f-not last-dtack-))
                           reg-bus
                           (v-wire data-in
                                   (vft-buf (f-not rw-)
                                            (v-threefix data-out))))))
                  (list* (f-buf (nth 3 (v-threefix last-pc-reg)))
                         (f-buf (nth 2 alu-bus))
                         (f-buf hdack-)
                         (ft-buf (f-buf hdack-) (f-buf rw-))
                         (ft-buf (f-buf hdack-) strobe-)
                         (append5
                          (vft-buf (f-buf hdack-) (v-threefix addr-out))
                          (vft-buf (f-not (f-buf rw-)) (v-threefix data-out))
                          (v-threefix flags)
                          (v-threefix state)
                          (v-threefix
                           (subrange (v-threefix i-reg) 28 31)))))))))))
  ;;Hint
  ((enable chip& chip*$destructure
           chip-module$value ;chip-module$state
           vdd-parametric$value
           ttl-clk-input$value ttl-input$value
           ttl-output$value ttl-tri-output$value
           ttl-output-fast$value ttl-tri-output-fast$value
           ttl-input-pads$value ttl-output-pads$value
           ttl-bidirect-pads$value ttl-tri-output-pads$value
           f-not-f-not=f-buf state)
   (disable indices *1*indices open-indices nth open-nth *1*nth
            make-tree *1*make-tree make-list *1*make-list
            threefix f-gates=b-gates open-v-threefix append-v-threefix)
   (disable-theory f-gates b-gates)))


;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    CHIP-SYSTEM
;;;
;;;    CHIP + MEMORY
;;;
;;; INPUTS:
;;;
;;;    CLK  -- The system clock
;;;
;;;    TI   -- Test input (Scan in)
;;;    TE   -- Test enable (Scan enable)
;;;
;;;    RESET-  --  RESET input
;;;    HOLD-   --  HOLD input
;;;
;;;    DISABLE-REGFILE-   --  Must be high to write enable the register
;;;                           file RAM.  During scan-in, DISABLE-REGFILE- is
;;;                           held low to avoid spurious writes to the RAM.
;;;
;;;    TEST-REGFILE-    --  Normally high.  When low, forces the register file
```

```
;;;                         output mux to pass data directly from the RAM.
;;;
;;;    PC-REG-IN[0..3]  --  The PC-REG inputs.
;;;
;;;
;;; OUTPUTS:
;;;
;;;    None.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(module-generator
 (chip-system*)
 'chip-system
 ;;  Inputs
 (list* 'clk 'ti 'te 'reset- 'hold-
        'disable-regfile- 'test-regfile-
        #i(pc-reg 0 4))
 ;;  Outputs
 ()
 ;;  Body
 (list
  ;;  The FM9001.
  (list 'fm9001
        (list* 'po 'to 'timing 'hdack- 'rw- 'strobe-
               (append #i(addr-out 0 32)
                       (append #i(fm9001-data 0 32)
                               (append #i(flags 0 4)
                                       (append #i(cntl-state 0 5)
                                               #i(i-reg 28 4))))))
        'chip
        (list* 'clk 'ti 'te 'dtack- 'reset- 'hold-
               'disable-regfile- 'test-regfile-
               (append #i(pc-reg 0 4)
                       #i(data-bus 0 32))))
  ;;  Memory control busses.
  '(pullup-rw- (rw-bus) pullup (rw-))

  '(pullup-strobe- (strobe-bus) pullup (strobe-))

  (list 'address
        #i(addr-bus 0 32)
        #i(v-pullup 32)
        #i(addr-out 0 32))

  ;;  Memory
  (list 'mem
        (list* 'dtack- #i(mem-data 0 32))
        'mem-32x32
        (list* 'rw-bus 'strobe-bus
               (append #i(addr-bus 0 32) #i(data-bus 0 32))))

  ;;  Data busses.
  (list 'data-wire
        #i(data-wire 0 32)
        #i(v-wire 32)
        (append #i(fm9001-data 0 32) #i(mem-data 0 32)))

  (list 'data
        #i(data-bus 0 32)
        #i(v-pullup 32)
        #i(data-wire 0 32)))

 ;;  State
 '(fm9001 mem))
```

```
(disable *1*chip-system*)


(defn chip-system& (netlist)
  (and (equal (lookup-module 'chip-system netlist)
              (chip-system*))
       (let ((netlist (delete-module 'chip-system netlist)))
         (and (chip& netlist)
              (pullup& netlist)
              (v-pullup& netlist 32)
              (mem-32x32& netlist)
              (v-wire& netlist 32)))))


(disable chip-system&)


(defn chip-system$netlist ()
  (cons (chip-system*)
        (union
         (pullup$netlist)
         (union
          (v-pullup$netlist 32)
          (union
           (mem-32x32$netlist)
           (union (v-wire$netlist 32)
                  (chip$netlist)))))))


(defn memory-state-invariant (mem-state)
  (let
    ((mem              (car mem-state))
     (mem-cntl         (cadr mem-state))
     (mem-clock        (caddr mem-state))
     (mem-count        (cadddr mem-state))
     (mem-dtack        (caddddr mem-state))
     (mem-last-rw-     (cadddddr mem-state))
     (mem-last-address (caddddddr mem-state))
     (mem-last-data    (cadddddddr mem-state)))
    (and (memory-properp 32 32 mem)
         (properp mem-last-address)
         (properp mem-last-data)
         (equal (length mem-last-data) 32))))


(defn chip-system-invariant (state)
  (and (machine-state-invariant (car state))
       (memory-state-invariant (cadr state))))
```

```
(prove-lemma equal-memory-value-for-chip-system$state (rewrite)
  (equal (memory-value state strobe rw- address (collect-value args alist))
         (memory-value state strobe rw- address
                       (make-list (length args) (x)))))
```

```
(prove-lemma chip-system$state-help (rewrite)
  (let
    ((regs (list regs-regs regs-we regs-data regs-addr)))
    (let
      ((machine-state
        (list
         regs flags a-reg b-reg i-reg data-out addr-out
         last-reset- last-dtack- last-hold- last-pc-reg
         cntl-state))
       (inputs
        (list* clk ti te reset- hold-
               disable-regfile- test-regfile-
               pc-reg-in)))
      (implies
       (and (chip-system& netlist)
            (chip-system-invariant (list machine-state mem-state))
            (equal te f)
            (equal disable-regfile- t)
            (equal test-regfile- t)
            (properp pc-reg-in) (equal (length pc-reg-in) 4))
       (equal (dual-eval 2 'chip-system
                         inputs (list machine-state mem-state) netlist)
              (let
                ((state           (state           (v-threefix cntl-state)))
                 (rw-             (rw-             (v-threefix cntl-state)))
                 (strobe-         (strobe-         (v-threefix cntl-state)))
                 (hdack-          (hdack-          (v-threefix cntl-state)))
                 (we-regs         (we-regs         (v-threefix cntl-state)))
                 (we-flags        (we-flags        (v-threefix cntl-state)))
                 (we-a-reg        (we-a-reg        (v-threefix cntl-state)))
                 (we-b-reg        (we-b-reg        (v-threefix cntl-state)))
                 (we-i-reg        (we-i-reg        (v-threefix cntl-state)))
                 (we-data-out     (we-data-out     (v-threefix cntl-state)))
                 (we-addr-out     (we-addr-out     (v-threefix cntl-state)))
                 (we-hold-        (we-hold-        (v-threefix cntl-state)))
                 (we-pc-reg       (we-pc-reg       (v-threefix cntl-state)))
                 (data-in-select  (data-in-select  (v-threefix cntl-state)))
                 (dec-addr-out    (dec-addr-out    (v-threefix cntl-state)))
                 (select-immediate (select-immediate (v-threefix cntl-state)))
                 (regs-address    (regs-address    (v-threefix cntl-state)))
                 (alu-c           (alu-c           (v-threefix cntl-state)))
                 (alu-op          (alu-op          (v-threefix cntl-state)))
                 (alu-zero        (alu-zero        (v-threefix cntl-state)))
                 (alu-mpg         (alu-mpg         (v-threefix cntl-state))))
                (let
                  ((ext-addr-out
                    (v-pullup (vft-buf (f-buf hdack-)
                                       (v-threefix addr-out))))
                   (ext-rw-
                    (f-pullup (ft-buf (f-buf hdack-) (f-buf  rw-))))
                   (ext-strobe-
                    (f-pullup (ft-buf (f-buf hdack-) strobe-)))
                   (ext-data-out
                    (vft-buf (f-not (f-buf rw-)) (v-threefix data-out))))
```

```
(let
  ((mem-response
    (memory-value mem-state ext-strobe- ext-rw-
                  ext-addr-out
                  (make-list 32 (x)))))
  (let
    ((dtack- (car mem-response))
     (ext-data-bus (v-pullup
                    (v-wire ext-data-out
                            (cdr mem-response)))))
    (let
      ((reg-bus (f$extend-immediate
                 select-immediate
                 (a-immediate (v-threefix i-reg))
                 (f$read-regs regs-address regs)))
       (alu-bus (f$core-alu alu-c
                            (v-threefix a-reg)
                            (v-threefix b-reg)
                            alu-zero alu-mpg alu-op
                            (make-tree 32)))
       (data-in (v-threefix
                 (v-wire ext-data-bus
                         ext-data-out))))
      (let
        ((addr-out-bus (f$dec-pass dec-addr-out reg-bus))
         (abi-bus
          (fv-if (f-nand data-in-select
                         (f-not last-dtack-))
                 reg-bus
                 data-in)))
        (list
         (list
          (f$write-regs we-regs regs-address regs
                        (bv alu-bus))
          (f$update-flags flags we-flags alu-bus)
          (fv-if we-a-reg abi-bus a-reg)
          (fv-if we-b-reg abi-bus b-reg)
          (fv-if we-i-reg abi-bus i-reg)
          (fv-if we-data-out (bv alu-bus) data-out)
          (fv-if we-addr-out addr-out-bus addr-out)
          (f-buf reset-)
          (f-or strobe- (f-buf dtack-))
          (f-if we-hold- (f-buf hold-) last-hold-)
          (fv-if we-pc-reg pc-reg-in last-pc-reg)
          (v-threefix (f$next-cntl-state
                       (f-buf last-reset-)
                       (f-buf last-dtack-)
                       (f-buf last-hold-)
                       rw-
                       state
                       (v-threefix i-reg)
                       (v-threefix flags)
                       (v-threefix last-pc-reg)
                       regs-address)))
         (next-memory-state
```

```
                           mem-state ext-strobe- ext-rw-
                           ext-addr-out
                           ext-data-bus))))))))))))
  ;;Hint
  ((enable chip-system& chip-system*$destructure
           chip$value chip$state
           mem-32x32$structured-value mem-32x32$structured-state
           pullup$value v-pullup$value v-wire$value)
   (disable indices *1*indices open-indices nth open-nth *1*nth
            make-tree *1*make-tree threefix f-gates=b-gates
            open-v-threefix append-v-threefix
            next-memory-state memory-value *1*make-list)
   (disable-theory f-gates b-gates)))


(disable chip-system$state-help)


(prove-lemma chip-system$state (rewrite)
  (let
    ((regs (list regs-regs regs-we regs-data regs-addr)))
    (let
      ((machine-state
        (list
         regs flags a-reg b-reg i-reg data-out addr-out
         last-reset- last-dtack- last-hold- last-pc-reg
         cntl-state))
       (inputs
        (list* clk ti te reset- hold-
               disable-regfile- test-regfile-
               pc-reg-in)))
      (implies
       (and (chip-system& netlist)
            (chip-system-invariant (list machine-state mem-state))
            (equal te f)
            (equal disable-regfile- t)
            (equal test-regfile- t)
            (properp pc-reg-in) (equal (length pc-reg-in) 4))
       (equal (dual-eval 2 'chip-system
                         inputs (list machine-state mem-state) netlist)
              (fm9001-next-state (list machine-state mem-state)
                                 (list* reset- hold- pc-reg-in))))))
  ;;Hint
  ((enable chip-system$state-help fm9001-next-state open-nth
           subrange-cons)
   (disable indices *1*indices open-indices nth *1*nth
            make-tree *1*make-tree threefix f-gates=b-gates
            open-v-threefix append-v-threefix
            next-memory-state memory-value *1*make-list
            machine-state-invariant memory-state-invariant)
   (enable-theory fm9001-hardware-state-accessors
                  fm9001-external-input-accessors)
   (disable-theory f-gates b-gates)))
```

```
(disable chip-system$state)

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    Induction proofs and specifications.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(prove-lemma next-memory-state-preserves-memory-invariant (rewrite)
  (implies
   (and (memory-state-invariant mem-state)
        (properp data)
        (equal (length data) 32)
        (equal (length address) 32))
   (memory-state-invariant
    (next-memory-state mem-state strobe- rw- address data)))
  ;;Hint
  ((enable next-memory-state)))


(prove-lemma memory-properp-dual-port-ram-state-crock (rewrite)
  (implies
   (and
    (memory-properp 4 32 regs)
    (equal (length a-address) 4)
    (equal (length b-address) 4)
    (properp data)
    (equal (length data) 32))
   (memory-properp
    4 32
    (dual-port-ram-state
     32 4
     (append a-address (append b-address (cons we data))) regs))))


(prove-lemma all-ramp-mem-dual-port-ram-state-crock (rewrite)
  (implies
   (and
    (all-ramp-mem 4 regs)
    (equal (length a-address) 4)
    (equal (length b-address) 4)
    (properp data)
    (equal (length data) 32))
   (all-ramp-mem
    4
    (dual-port-ram-state
     32 4
     (append a-address (append b-address (cons we data))) regs))))
```

```
(prove-lemma fm9001-preserves-chip-system-invariant (rewrite)
  (implies
   (and (chip-system-invariant state)
        (properp (pc-reg-input inputs))
        (equal (length (pc-reg-input inputs)) 4))
   (chip-system-invariant
    (fm9001-next-state state inputs)))
  ;;Hint
  ((enable fm9001-next-state f$write-regs regs-address cntl-state open-nth)
   (disable *1*make-tree *1*make-list open-v-threefix threefix
            dual-port-ram-state)
   (enable-theory fm9001-hardware-state-accessors)
   (disable-theory f-gates)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    CHIP-SYSTEM-OPERATING-INPUTS-P
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn chip-system-input-invariant (inputs)
  (let
    ((clk              (car inputs))
     (ti               (cadr inputs))
     (te               (caddr inputs))
     (reset-           (cadddr inputs))
     (hold-            (caddddr inputs))
     (disable-regfile- (cadddddr inputs))
     (test-regfile-    (caddddddr inputs))
     (pc-reg-in        (cddddddddr inputs)))
    (and (equal te f)
         (equal disable-regfile- t)
         (equal test-regfile- t)
         (properp pc-reg-in)
         (equal (length pc-reg-in) 4))))
```

```
(prove-lemma rewrite-chip-system-input-invariant ()
  (equal (chip-system-input-invariant inputs)
         (let
           ((clk             (car inputs))
            (ti              (cadr inputs))
            (te              (caddr inputs))
            (reset-          (cadddr inputs))
            (hold-           (caddddr inputs))
            (disable-regfile- (cadddddr inputs))
            (test-regfile-   (caddddddr inputs))
            (pc-reg-in       (cdddddddr inputs)))
           (and (equal inputs
                       (list* clk ti te reset- hold-
                              disable-regfile- test-regfile-
                              pc-reg-in))
                (equal te f)
                (equal disable-regfile- t)
                (equal test-regfile- t)
                (properp pc-reg-in)
                (equal (length pc-reg-in) 4)))))


(defn chip-system-operating-inputs-p (inputs n)
  (if (zerop n)
      t
    (let
      ((clk             (caar inputs))
       (ti              (cadar inputs))
       (te              (caddar inputs))
       (reset-          (cadddar inputs))
       (hold-           (caddddar inputs))
       (disable-regfile- (cadddddar inputs))
       (test-regfile-   (caddddddar inputs))
       (pc-reg-in       (cdddddddar inputs)))
      (and (chip-system-input-invariant (car inputs))
           (chip-system-operating-inputs-p (cdr inputs) (sub1 n))))))


(disable chip-system-operating-inputs-p)
```

```
(prove-lemma open-chip-system-operating-inputs-p (rewrite)
  (and
   (implies
    (zerop n)
    (equal (chip-system-operating-inputs-p inputs n)
           t))
   (implies
    (not (zerop n))
    (equal (chip-system-operating-inputs-p inputs n)
           (and (chip-system-input-invariant (car inputs))
                (chip-system-operating-inputs-p (cdr inputs) (sub1 n))))))
  ;;Hint
  ((enable chip-system-operating-inputs-p)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    MAP-UP-1-INPUT and MAP-UP-INPUTS
;;;
;;;    Convert low-level input streams to mid-level input streams.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn map-up-1-input (inputs)
  (let
    ((clk              (car inputs))
     (ti               (cadr inputs))
     (te               (caddr inputs))
     (reset-           (cadddr inputs))
     (hold-            (caddddr inputs))
     (disable-regfile- (cadddddr inputs))
     (test-regfile-    (caddddddr inputs))
     (pc-reg-in        (cdddddddr inputs)))
    (list* reset- hold- pc-reg-in)))


(defn map-up-inputs (inputs)
  (if (nlistp inputs)
      nil
    (cons (map-up-1-input (car inputs))
          (map-up-inputs (cdr inputs)))))


(disable map-up-inputs)
```

```
(prove-lemma open-map-up-inputs (rewrite)
  (and
   (implies
    (nlistp inputs)
    (equal (map-up-inputs inputs)
           nil))
   (implies
    (listp inputs)
    (equal (map-up-inputs inputs)
           (cons (map-up-1-input (car inputs))
                 (map-up-inputs (cdr inputs))))))
  ;;Hint
  ((enable map-up-inputs)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    The proofs that the DUAL-EVAL model of the processor equal
;;;    RUN-FM9001.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(prove-lemma chip-system=fm9001$step (rewrite)
  (implies
   (and (chip-system& netlist)
        (fm9001-state-structure state)
        (chip-system-invariant state)
        (chip-system-input-invariant inputs))
   (equal (dual-eval 2 'chip-system inputs state netlist)
          (fm9001-next-state state (map-up-1-input inputs))))
  ;;hint
  ((disable chip-system-invariant fm9001-state-structure
            chip-system-input-invariant
            car-cdr-elim cons-car-cdr)
   (use (chip-system$state
         (regs-regs       (caaar state))
         (regs-we         (cadaar state))
         (regs-data       (caddaar state))
         (regs-addr       (cadddaar state))
         (flags           (cadar state))
         (a-reg           (caddar state))
         (b-reg           (cadddar state))
         (i-reg           (caddddar state))
         (data-out        (cadddddar state))
         (addr-out        (caddddddar state))
         (last-reset-     (cadddddddar state))
         (last-dtack-     (caddddddddar state))
         (last-hold-      (cadddddddddar state))
         (last-pc-reg     (caddddddddddar state))
         (cntl-state      (cadddddddddddar state))
         (mem-state       (list (caadr state)
                                (cadadr state)
                                (caddadr state)
                                (cadddadr state)
                                (caddddadr state)
                                (cadddddadr state)
                                (caddddddadr state)
                                (cadddddddadr state)))
         (clk             (car inputs))
         (ti              (cadr inputs))
         (te              (caddr inputs))
         (reset-          (cadddr inputs))
         (hold-           (caddddr inputs))
         (disable-regfile- (cadddddr inputs))
         (test-regfile-   (caddddddr inputs))
         (pc-reg-in       (cddddddddr inputs)))
        (fm9001-state-as-a-list)
        (rewrite-chip-system-input-invariant))))


(disable chip-system=fm9001$step)
```

```
(defn chip-system=run-fm9001$induction (state inputs fm9001-inputs n)
  (if (zerop n)
      t
    (chip-system=run-fm9001$induction
     (fm9001-next-state state (car fm9001-inputs))
     (cdr inputs) (cdr fm9001-inputs) (sub1 n))))


(prove-lemma chip-system=run-fm9001 (rewrite)
  (implies
   (and (chip-system& netlist)
        (fm9001-state-structure state)
        (chip-system-invariant state)
        (chip-system-operating-inputs-p inputs n)
        (equal fm9001-inputs (map-up-inputs inputs)))
   (equal (simulate-dual-eval-2 'chip-system inputs state netlist n)
          (run-fm9001 state fm9001-inputs n)))
  ;;Hint
  ((induct (chip-system=run-fm9001$induction state inputs fm9001-inputs n))
   (enable chip-system=fm9001$step pc-reg-input open-subrange
           run-fm9001-step-case)
   (disable fm9001-state-structure chip-system-invariant
            chip-system-input-invariant map-up-1-input)))
```

14.65 "expand-fm9001.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    EXPAND-FM9001.EVENTS
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;   RUN-INPUTS-P inputs n
;;;
;;;   This predicate recognizes input streams that don't cause the machine to
;;;   enter the RESET state.


(defn run-inputs-p (inputs n)
  (if (zerop n)
      t
    (and (equal (reset--input (car inputs)) t)
         (properp (pc-reg-input (car inputs)))
         (equal (length (pc-reg-input (car inputs))) 4)
         (run-inputs-p (cdr inputs) (sub1 n)))))


(disable run-inputs-p)


(prove-lemma open-run-inputs-p (rewrite)
  (and
   (implies
    (zerop n)
    (equal (run-inputs-p inputs n)
           t))
   (implies
    (not (zerop n))
    (equal (run-inputs-p inputs n)
           (and (equal (reset--input (car inputs)) t)
                (properp (pc-reg-input (car inputs)))
                (equal (length (pc-reg-input (car inputs))) 4)
                (run-inputs-p (cdr inputs) (sub1 n))))))
  ;;Hint
  ((enable run-inputs-p)))


(prove-lemma run-inputs-p-plus (rewrite)
  (equal (run-inputs-p inputs (plus n m))
         (and (run-inputs-p inputs n)
              (run-inputs-p (nthcdr n inputs) m)))
  ;;Hint
  ((enable run-inputs-p nthcdr plus)))
```

```
(prove-lemma run-inputs-p-1 (rewrite)
  (implies
   (and (run-inputs-p inputs n)
        (not (zerop n)))
   (run-inputs-p inputs 1))
  ;;Hint
  ((enable run-inputs-p)))


(prove-lemma open-run-inputs-p-add1 (rewrite)
   (equal (run-inputs-p inputs (add1 n))
          (and (equal (reset--input (car inputs)) t)
               (properp (pc-reg-input (car inputs)))
               (equal (length (pc-reg-input (car inputs))) 4)
               (run-inputs-p (cdr inputs) n)))
  ;;Hint
  ((enable run-inputs-p)))

;;;  REGFILE-OKP regs
;;;
;;;  This predicate states that the register file RAM, and the address, data,
;;;  and write-enable latches are Boolean and the proper size.


(defn regfile-okp (regs)
  (and (equal (length regs) 4)
       (memory-okp 4 32 (car regs))
       (all-ramp-mem 4 (car regs))
       (boolp (cadr regs))
       (and (bvp (caddr regs)) (equal (length (caddr regs)) 32))
       (and (bvp (cadddr regs)) (equal (length (cadddr regs)) 4))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    LOW LEVEL EXPANSION
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(prove-lemma rewrite-fm9001-next-state-for-step-lemmas (rewrite)
  (implies
   (and (bvp flags)
        (bvp a-reg)
        (bvp b-reg)
        (bvp i-reg)
        (bvp data-out)
        (bvp addr-out)
        (boolp last-reset-)
        (boolp last-dtack-)
        (boolp last-hold-)
        (bvp last-pc-reg)
        (bvp cntl-state))
  (equal
   (fm9001-next-state
    (list (list regs
                flags
                a-reg b-reg i-reg
                data-out addr-out
                last-reset- last-dtack- last-hold-
                last-pc-reg cntl-state)
          mem-state)
    external-inputs)
   (let
     ((reset-    (reset--input external-inputs))
      (hold-     (hold--input  external-inputs))
      (pc-reg-in (pc-reg-input external-inputs))

     (state            (state            cntl-state))
     (rw-              (rw-              cntl-state))
     (strobe-          (strobe-          cntl-state))
     (hdack-           (hdack-           cntl-state))
     (we-regs          (we-regs          cntl-state))
     (we-flags         (we-flags         cntl-state))
     (we-a-reg         (we-a-reg         cntl-state))
     (we-b-reg         (we-b-reg         cntl-state))
     (we-i-reg         (we-i-reg         cntl-state))
     (we-data-out      (we-data-out      cntl-state))
     (we-addr-out      (we-addr-out      cntl-state))
     (we-hold-         (we-hold-         cntl-state))
     (we-pc-reg        (we-pc-reg        cntl-state))
     (data-in-select   (data-in-select   cntl-state))
     (dec-addr-out     (dec-addr-out     cntl-state))
     (select-immediate (select-immediate cntl-state))
     (regs-address     (regs-address     cntl-state))
     (alu-c            (alu-c            cntl-state))
     (alu-op           (alu-op           cntl-state))
     (alu-zero         (alu-zero         cntl-state))
     (alu-mpg          (alu-mpg          cntl-state)))
   (let
     ((ext-addr-out
       (v-pullup (vft-buf (f-buf hdack-)
                          addr-out)))
      (ext-rw-
```

```
      (f-pullup (ft-buf (f-buf hdack-) (f-buf  rw-))))
     (ext-strobe-
      (f-pullup (ft-buf (f-buf hdack-) strobe-)))
     (ext-data-out
      (vft-buf (f-not (f-buf rw-)) data-out)))
    (let
      ((mem-response
        (memory-value mem-state ext-strobe- ext-rw-
                      ext-addr-out
                      (make-list 32 (x)))))
      (let
        ((reg-bus (f$extend-immediate
                   select-immediate
                   (a-immediate i-reg)
                   (f$read-regs regs-address regs)))
         (alu-bus (f$core-alu alu-c
                              a-reg
                              b-reg
                              alu-zero alu-mpg alu-op
                              (make-tree 32)))
         (dtack- (car mem-response))
         (data-in (v-threefix
                   (v-wire (v-pullup
                            (v-wire ext-data-out
                                    (cdr mem-response)))
                           ext-data-out))))
        (let
          ((addr-out-bus (f$dec-pass dec-addr-out reg-bus))
           (abi-bus
            (fv-if (f-nand data-in-select
                           (f-not last-dtack-))
                   reg-bus
                   data-in)))
          (list
           (list
            (f$write-regs we-regs regs-address regs (bv alu-bus))
            (f$update-flags flags we-flags alu-bus)
            (fv-if we-a-reg abi-bus a-reg)
            (fv-if we-b-reg abi-bus b-reg)
            (fv-if we-i-reg abi-bus i-reg)
            (fv-if we-data-out (bv alu-bus) data-out)
            (fv-if we-addr-out addr-out-bus addr-out)
            (f-buf reset-)
            (f-or strobe- (f-buf dtack-))
            (f-if we-hold- (f-buf hold-) last-hold-)
            (fv-if we-pc-reg pc-reg-in last-pc-reg)
            (v-threefix (f$next-cntl-state
                         last-reset-
                         last-dtack-
                         last-hold-
                         rw-
                         state
                         i-reg
                         flags
                         last-pc-reg
```

```
                        regs-address)))
             (next-memory-state
              mem-state ext-strobe- ext-rw-
              ext-addr-out
              (v-pullup
               (v-wire ext-data-out
                       (cdr mem-response))))))))))))
  ;;Hint
  ((enable fm9001-next-state boolp-b-gates open-nth)
   (enable-theory fm9001-hardware-state-accessors)
   (disable *1*make-list)
   (disable-theory f-gates)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    SINGLE STEP SIMULATIONS
;;;
;;;  The STEP-FM9001 macro creates a lemma that describes the new state
;;;  computed by a single step of the low-level machine from a given initial
;;;  state.  This procedure is complicated by the fact that the next state
;;;  depends on the state of the memory.  Rather than produce a single lemma
;;;  that captures the behavior for all possible memory states, we create
;;;  separate lemmas only for those processor-state/memory-state configurations
;;;  that are possible during normal instruction execution.  This is important
;;;  because if the memory protocol is not followed, then unknown values may be
;;;  loaded into the processor registers.  What we are doing here is creating
;;;  a state-by-state, almost-Boolean specification of the low-level machine.
;;;  So we obviously we want to insure that the processor state remains
;;;  Boolean.
;;;
;;;  STEP-FM9001 is defined in "expand-fm9001.lisp".
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;  Page numbers refer to the page numbers of the state diagram drawings,
;;;  e.g., Page 0 is the figure with caption ``States Figure 0'' in
;;;  Technical Report 86.

;;;  Page 0


(step-fm9001 fetch0 :suffix 0)

(step-fm9001 fetch0 :suffix 1 :mem-state 1)

(step-fm9001 fetch0 :suffix 2 :mem-state 2
             :addr-stable t :data-stable t :last-rw- f
             :hyps (zerop mem-count))


(step-fm9001 fetch1 :hyps (boolp last-rw-) :last-rw- last-rw-)

(step-fm9001 fetch2 :addr-stable t :split-term (zerop (car clock)))
```

```
(step-fm9001 fetch3 :suffix 0
             :addr-stable t :dtack- f :mem-state 1 :mem-dtack t
             :hyps (zerop mem-count))

(step-fm9001 fetch3 :suffix 1 :addr-stable t :dtack- t :mem-state 1
             :mem-dtack f :split-term (zerop mem-count))


(step-fm9001 decode :mem-state 1)

;;;  Page 1


(step-fm9001 rega)

(step-fm9001 regb)

(step-fm9001 update :mem-state mem-state
               :hyps (or (equal mem-state 0) (equal mem-state 1)))

;;;  Page 2


(step-fm9001 reada0)

(step-fm9001 reada1)

(step-fm9001 reada2 :addr-stable t :split-term (zerop (car clock)))


(step-fm9001 reada3 :suffix 0
               :addr-stable t :dtack- f :mem-state 1 :mem-dtack t
               :hyps (zerop mem-count))

(step-fm9001 reada3 :suffix 1 :addr-stable t :dtack- t :mem-state 1
             :mem-dtack f :split-term (zerop mem-count))

;;;  Page 3


(step-fm9001 readb0 :mem-state mem-state
               :hyps (or (equal mem-state 0) (equal mem-state 1)))

(step-fm9001 readb1)

(step-fm9001 readb2 :addr-stable t :split-term (zerop (car clock)))


(step-fm9001 readb3 :suffix 0
               :addr-stable t :dtack- f :mem-state 1 :mem-dtack t
               :hyps (zerop mem-count))
```

```
(step-fm9001 readb3 :suffix 1 :addr-stable t :dtack- t :mem-state 1
             :mem-dtack f :split-term (zerop mem-count))

;;;  Page 4


(step-fm9001 write0 :mem-state mem-state
               :hyps (or (equal mem-state 0) (equal mem-state 1)))

(step-fm9001 write1)

(step-fm9001 write2 :last-rw- f :addr-stable t :data-stable t
             :split-term (zerop (car clock)))


(step-fm9001 write3 :suffix 0
               :addr-stable t :data-stable t :dtack- f :mem-state 2
               :mem-dtack t :last-rw- f
               :hyps (zerop mem-count))

(step-fm9001 write3 :suffix 1 :addr-stable t :data-stable t :dtack- t
             :mem-state 2 :mem-dtack f :last-rw- f
             :split-term (zerop mem-count))

;;;  Page 5


(step-fm9001 sefa0)

(step-fm9001 sefa1)

(step-fm9001 sefb0)

(step-fm9001 sefb1)



(deftheory fm9001-step-theory
  (sefb1$step sefb0$step sefa1$step sefa0$step write3$step1 write3$step0
              write2$step write1$step write0$step readb3$step1 readb3$step0
              readb2$step readb1$step readb0$step reada3$step1 reada3$step0
              reada2$step reada1$step reada0$step update$step regb$step
              rega$step decode$step fetch3$step1 fetch3$step0 fetch2$step
              fetch1$step fetch0$step2 fetch0$step1 fetch0$step0))


#|
;;;   Page 6

It was never necessary to expand these states.


(step-fm9001 hold0)
```

```
(step-fm9001 hold1)


(step-fm9001 reset0)

(step-fm9001 reset1)

(step-fm9001 reset2)

|#

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    Inductive progress proofs
;;;
;;;   For each of the states that wait for memory, we must prove that the
;;;   processor will eventually move on.  The lemmas below tell how long the
;;;   processor will remain in a given state, before the internal registers are
;;;   in such a condition that the machine will leave that state on the next
;;;   clock cycle.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;    READ-FN reg clock n
;;;
;;;    READ-FN gives the value produced by the ABI-MUX as we wait for DTACK.


(defn read-fn (reg0 reg1 n)
  (if (zerop n)
      reg0
    (read-fn reg1 reg1 (sub1 n))))


(disable read-fn)


(prove-lemma open-read-fn (rewrite)
  (and
   (implies
    (zerop n)
    (equal (read-fn reg0 reg1 n)
           reg0))
   (implies
    (not (zerop n))
    (equal (read-fn reg0 reg1 n)
           (read-fn reg1 reg1 (sub1 n)))))
  ;;Hint
  ((enable read-fn)))
```

```
(prove-lemma length-read-fn (rewrite)
  (implies
   (and (equal (length reg0) 32)
        (equal (length reg1) 32))
   (equal (length (read-fn reg0 reg1 n)) 32))
  ;;Hint
  ((enable read-fn)))


(prove-lemma properp-read-fn (rewrite)
  (implies
   (and (properp reg0)
        (properp reg1))
   (properp (read-fn reg0 reg1 n)))
  ;;Hint
  ((enable read-fn)))


(prove-lemma bvp-read-fn (rewrite)
  (implies
   (and (bvp reg0)
        (bvp reg1))
   (bvp (read-fn reg0 reg1 n)))
  ;;Hint
  ((enable read-fn)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    FETCH3
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn fetch3$induction
  (n clock count inputs
     regs a-reg b-reg
     rw- regs-address i-reg flags pc-reg
     last-rw- last-regs-address last-i-reg last-flags
     last-data)

  (if (zerop n)
      t
    (fetch3$induction
     (sub1 n) clock (sub1 count) (cdr inputs)
     (write-regs f 0 regs 0) a-reg b-reg
     t regs-address (read-regs pc-reg regs) flags pc-reg
     rw- pc-reg i-reg flags
     (v-threefix last-data))))
```

```
(prove-lemma fetch3$progress-help ()
  (implies
   (and (cv-hyps t last-regs-address last-i-reg last-flags pc-reg)
        (cv-hyps t last-regs-address i-reg flags pc-reg)
        (memory-okp 32 32 mem)
        (regfile-okp regs)
        (bvp a-reg)
        (equal (length a-reg) 32)
        (bvp b-reg)
        (equal (length b-reg) 32)
        (bvp addr-out)
        (equal (length addr-out) 32)
        (bvp data-out)
        (equal (length data-out) 32)
        (boolp hold-)
        (run-inputs-p inputs (add1 n))
        (leq n count))
   (equal
    (run-fm9001
     (list
      (list regs flags
            a-reg b-reg i-reg
            data-out addr-out
            t t hold-
            pc-reg
            (cv_fetch3 t last-regs-address last-i-reg
                       last-flags pc-reg))
      (list mem 1 clock count f t addr-out last-data))
     inputs
     (add1 n))
    (list
     (list (write-regs f 0 regs 0) flags
           a-reg b-reg (read-fn i-reg (read-regs pc-reg regs) (add1 n))
           data-out addr-out
           t (not (leq count n)) hold-
           pc-reg
           (cv_fetch3 t pc-reg
                      (read-fn i-reg (read-regs pc-reg regs) n) flags pc-reg))
     (list mem 1 clock (difference count (add1 n))
            (leq count n) t addr-out (v-threefix last-data)))))
  ;;hint
  ((induct (fetch3$induction
            n clock count inputs
            regs a-reg b-reg
            rw- regs-address i-reg flags pc-reg
            last-rw- last-regs-address last-i-reg last-flags
            last-data))
   (enable plus nthcdr difference open-read-fn write-regs-ok)))
```

```
(prove-lemma fetch3$progress (rewrite)
  (implies
   (and (cv-hyps t last-regs-address last-i-reg last-flags pc-reg)
        (cv-hyps t last-regs-address i-reg flags pc-reg)
        (memory-okp 32 32 mem)
        (regfile-okp regs)
        (bvp a-reg)
        (equal (length a-reg) 32)
        (bvp b-reg)
        (equal (length b-reg) 32)
        (bvp addr-out)
        (equal (length addr-out) 32)
        (bvp data-out)
        (equal (length data-out) 32)
        (boolp hold-)
        (run-inputs-p inputs count)
        (not (zerop count)))
   (equal
    (run-fm9001
     (list
      (list regs flags
            a-reg b-reg i-reg
            data-out addr-out
            t t hold-
            pc-reg
            (cv_fetch3 t last-regs-address last-i-reg last-flags pc-reg))
      (list mem 1 clock (sub1 count) f t addr-out last-data))
     inputs
     count)
    (list
     (list (write-regs f 0 regs 0) flags
           a-reg b-reg (read-fn i-reg (read-regs pc-reg regs) count)
           data-out addr-out
           t f hold-
           pc-reg
           (cv_fetch3 t pc-reg
                      (read-fn i-reg (read-regs pc-reg regs) (sub1 count))
                      flags
                      pc-reg))
     (list mem 1 clock 0 t t addr-out (v-threefix last-data)))))
  ;;hint
  ((use (fetch3$progress-help (count (sub1 count)) (n (sub1 count))))
   (disable plus-add1 open-run-inputs-p)
   (enable open-read-fn)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    READA3
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defn reada3$induction (n clock count inputs
                          regs a-reg b-reg
                          rw- regs-address i-reg flags pc-reg
                          last-rw- last-regs-address last-i-reg last-flags
                          last-data)

  (if (zerop n)
      t
    (reada3$induction
     (sub1 n) clock (sub1 count) (cdr inputs)
     (write-regs f 0 regs 0) (read-regs pc-reg regs) b-reg
     t regs-address i-reg flags pc-reg
     rw- pc-reg i-reg flags
     (v-threefix last-data))))
```

```
(prove-lemma reada3$progress-help ()
  (implies
   (and (cv-hyps t last-regs-address last-i-reg last-flags pc-reg)
        (cv-hyps t last-regs-address i-reg flags pc-reg)
        (memory-okp 32 32 mem)
        (regfile-okp regs)
        (bvp a-reg)
        (equal (length a-reg) 32)
        (bvp b-reg)
        (equal (length b-reg) 32)
        (bvp addr-out)
        (equal (length addr-out) 32)
        (bvp data-out)
        (equal (length data-out) 32)
        (boolp hold-)
        (run-inputs-p inputs (add1 n))
        (leq n count))
   (equal
    (run-fm9001
     (list
      (list regs flags
            a-reg b-reg i-reg
            data-out addr-out
            t t hold-
            pc-reg
            (cv_reada3 t last-regs-address last-i-reg
                       last-flags pc-reg))
      (list mem 1 clock count f t addr-out last-data))
     inputs
     (add1 n))
    (list
     (list (write-regs f 0 regs 0) flags
           (read-fn a-reg (read-regs pc-reg regs) (add1 n)) b-reg i-reg
           data-out addr-out
           t (not (leq count n)) hold-
           pc-reg
           (cv_reada3 t pc-reg i-reg flags pc-reg))
     (list mem 1 clock (difference count (add1 n))
           (leq count n) t addr-out (v-threefix last-data)))))
  ;;hint
  ((induct (reada3$induction
            n clock count inputs
            regs a-reg b-reg
            rw- regs-address i-reg flags pc-reg
            last-rw- last-regs-address last-i-reg last-flags
            last-data))
   (enable plus nthcdr difference open-read-fn run-inputs-p
           write-regs-ok)
   (disable open-run-inputs-p)))
```

```
(prove-lemma reada3$progress (rewrite)
  (implies
   (and (cv-hyps t last-regs-address last-i-reg last-flags pc-reg)
        (cv-hyps t last-regs-address i-reg flags pc-reg)
        (memory-okp 32 32 mem)
        (regfile-okp regs)
        (bvp a-reg)
        (equal (length a-reg) 32)
        (bvp b-reg)
        (equal (length b-reg) 32)
        (bvp addr-out)
        (equal (length addr-out) 32)
        (bvp data-out)
        (equal (length data-out) 32)
        (boolp hold-)
        (run-inputs-p inputs count)
        (not (zerop count)))
   (equal
    (run-fm9001
     (list
      (list regs flags
            a-reg b-reg i-reg
            data-out addr-out
            t t hold-
            pc-reg
            (cv_reada3 t last-regs-address last-i-reg last-flags pc-reg))
      (list mem 1 clock (sub1 count) f t addr-out last-data))
     inputs
     count)
    (list
     (list (write-regs f 0 regs 0) flags
           (read-fn a-reg (read-regs pc-reg regs) count) b-reg i-reg
           data-out addr-out
           t f hold-
           pc-reg
           (cv_reada3 t pc-reg i-reg flags pc-reg))
     (list mem 1 clock 0 t t addr-out (v-threefix last-data)))))
  ;;hint
  ((use (reada3$progress-help (count (sub1 count)) (n (sub1 count))))
   (disable plus-add1 open-run-inputs-p)
   (enable open-read-fn)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    READB3
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defn readb3$induction (n clock count inputs
                          regs a-reg b-reg
                          rw- regs-address i-reg flags pc-reg
                          last-rw- last-regs-address last-i-reg last-flags
                          last-data)

  (if (zerop n)
      t
    (readb3$induction
     (sub1 n) clock (sub1 count) (cdr inputs)
     (write-regs f 0 regs 0) a-reg (read-regs pc-reg regs)
     t regs-address i-reg flags pc-reg
     rw- pc-reg i-reg flags
     (v-threefix last-data))))
```

```
(prove-lemma readb3$progress-help ()
  (implies
   (and (cv-hyps t last-regs-address last-i-reg last-flags pc-reg)
        (cv-hyps t last-regs-address i-reg flags pc-reg)
        (memory-okp 32 32 mem)
        (regfile-okp regs)
        (bvp a-reg)
        (equal (length a-reg) 32)
        (bvp b-reg)
        (equal (length b-reg) 32)
        (bvp addr-out)
        (equal (length addr-out) 32)
        (bvp data-out)
        (equal (length data-out) 32)
        (boolp hold-)
        (run-inputs-p inputs (add1 n))
        (leq n count))
   (equal
    (run-fm9001
     (list
      (list regs flags
            a-reg b-reg i-reg
            data-out addr-out
            t t hold-
            pc-reg
            (cv_readb3 t last-regs-address last-i-reg
                       last-flags pc-reg))
      (list mem 1 clock count f t addr-out last-data))
     inputs
     (add1 n))
    (list
     (list (write-regs f 0 regs 0) flags
           a-reg (read-fn b-reg (read-regs pc-reg regs) (add1 n)) i-reg
           data-out addr-out
           t (not (leq count n)) hold-
           pc-reg
           (cv_readb3 t pc-reg i-reg flags pc-reg))
     (list mem 1 clock (difference count (add1 n))
           (leq count n) t addr-out (v-threefix last-data)))))
  ;;hint
  ((induct (readb3$induction
            n clock count inputs
            regs a-reg b-reg
            rw- regs-address i-reg flags pc-reg
            last-rw- last-regs-address last-i-reg last-flags
            last-data))
   (enable plus nthcdr difference open-read-fn run-inputs-p
           write-regs-ok)
   (disable open-run-inputs-p)))
```

```
(prove-lemma readb3$progress (rewrite)
  (implies
   (and (cv-hyps t last-regs-address last-i-reg last-flags pc-reg)
        (cv-hyps t last-regs-address i-reg flags pc-reg)
        (memory-okp 32 32 mem)
        (regfile-okp regs)
        (bvp a-reg)
        (equal (length a-reg) 32)
        (bvp b-reg)
        (equal (length b-reg) 32)
        (bvp addr-out)
        (equal (length addr-out) 32)
        (bvp data-out)
        (equal (length data-out) 32)
        (boolp hold-)
        (run-inputs-p inputs count)
        (not (zerop count)))
   (equal
    (run-fm9001
     (list
      (list regs flags
            a-reg b-reg i-reg
            data-out addr-out
            t t hold-
            pc-reg
            (cv_readb3 t last-regs-address last-i-reg last-flags pc-reg))
      (list mem 1 clock (sub1 count) f t addr-out last-data))
     inputs
     count)
    (list
     (list (write-regs f 0 regs 0) flags
           a-reg (read-fn b-reg (read-regs pc-reg regs) count) i-reg
           data-out addr-out
           t f hold-
           pc-reg
           (cv_readb3 t pc-reg i-reg flags pc-reg))
     (list mem 1 clock 0 t t addr-out (v-threefix last-data)))))
  ;;hint
  ((use (readb3$progress-help (count (sub1 count)) (n (sub1 count))))
   (disable plus-add1 open-run-inputs-p)
   (enable open-read-fn)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    WRITE3
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defn write3$induction
  (n clock count inputs
     regs a-reg b-reg
     rw- regs-address i-reg flags pc-reg
     last-rw- last-regs-address last-i-reg last-flags
     last-data)

  (if (zerop n)
      t
    (write3$induction
     (sub1 n) clock (sub1 count) (cdr inputs)
     (write-regs f 0 regs 0) a-reg b-reg
     f regs-address i-reg flags pc-reg
     rw- pc-reg i-reg flags
     (v-threefix last-data))))
```

```
(prove-lemma write3$progress-help ()
  (implies
   (and (cv-hyps t last-regs-address last-i-reg last-flags pc-reg)
        (cv-hyps t last-regs-address i-reg flags pc-reg)
        (memory-okp 32 32 mem)
        (regfile-okp regs)
        (bvp a-reg)
        (equal (length a-reg) 32)
        (bvp b-reg)
        (equal (length b-reg) 32)
        (bvp addr-out)
        (equal (length addr-out) 32)
        (bvp data-out)
        (equal (length data-out) 32)
        (boolp hold-)
        (run-inputs-p inputs (add1 n))
        (leq n count))
   (equal
    (run-fm9001
     (list
      (list regs flags
            a-reg b-reg i-reg
            data-out addr-out
            t t hold-
            pc-reg
            (cv_write3 f last-regs-address last-i-reg
                       last-flags pc-reg))
      (list mem 2 clock count f f addr-out data-out))
     inputs
     (add1 n))
    (list
     (list (write-regs f 0 regs 0) flags
           a-reg b-reg i-reg
           data-out addr-out
           t (not (leq count n)) hold-
           pc-reg
           (cv_write3 f pc-reg i-reg flags pc-reg))
     (list mem 2 clock (difference count (add1 n))
           (leq count n) f addr-out data-out))))
  ;;hint
  ((induct (write3$induction
            n clock count inputs
            regs a-reg b-reg
            rw- regs-address i-reg flags pc-reg
            last-rw- last-regs-address last-i-reg last-flags
            last-data))
   (enable plus nthcdr difference open-read-fn run-inputs-p
           write-regs-ok)
   (disable open-run-inputs-p)))
```

```
(prove-lemma write3$progress (rewrite)
  (implies
   (and (cv-hyps t last-regs-address last-i-reg last-flags pc-reg)
        (cv-hyps t last-regs-address i-reg flags pc-reg)
        (memory-okp 32 32 mem)
        (regfile-okp regs)
        (bvp a-reg)
        (equal (length a-reg) 32)
        (bvp b-reg)
        (equal (length b-reg) 32)
        (bvp addr-out)
        (equal (length addr-out) 32)
        (bvp data-out)
        (equal (length data-out) 32)
        (boolp hold-)
        (run-inputs-p inputs count)
        (not (zerop count)))
   (equal
    (run-fm9001
     (list
      (list regs flags
            a-reg b-reg i-reg
            data-out addr-out
            t t hold-
            pc-reg
            (cv_write3 f last-regs-address last-i-reg
                       last-flags pc-reg))
      (list mem 2 clock (sub1 count) f f addr-out data-out))
     inputs
     count)
    (list
     (list (write-regs f 0 regs 0) flags
           a-reg b-reg i-reg
           data-out addr-out
           t f hold-
           pc-reg
           (cv_write3 f pc-reg i-reg flags pc-reg))
     (list mem 2 clock 0 t f addr-out data-out))))
  ;;hint
  ((use (write3$progress-help (count (sub1 count)) (n (sub1 count))))
   (disable plus-add1 open-run-inputs-p)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    Chunk Simulations
;;;
;;;  SIM-FM9001 is a macro that creates lemmas that describe the operation of
;;;  the machine for multiple clock cycles.  We create a multi-state simulation
;;;  for each non-branching segment of the state diagrams.  For this purpose,
;;;  we consider the states that wait for memory as "non-branching".  Notice
;;;  that some of the "multi-state" lemmas only include a single state.
;;;
;;;  SIM-FM9001 is defined in "expand-fm9001.lisp"
;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;  ZEROP-NOT-ZEROP-CASES is a bogus recursive function used to force the
;;;  prover to do these multi-step simulations by cases.


(defn zerop-not-zerop-cases (n)
  (if (zerop n)
      t
    (if (not (zerop n))
        t
      (zerop-not-zerop-cases n)))
  ;;Hint
  ((lessp (count n))))


(sim-fm9001 fetch0 fetch0 :suffix 0)

(sim-fm9001 fetch0 fetch0 :suffix 1 :mem-state 1)

(sim-fm9001 fetch0 fetch0 :suffix 2 :mem-state 2
            :hyps (zerop mem-count)
            :addr-stable t :data-stable t :last-rw- f)


(sim-fm9001 fetch1 decode :hyps (and (equal hold- t) (boolp last-rw-))
            :last-rw- last-rw-
            :dtack-wait #.(w_fetch1->decode) :disable (fetch3$step1))


(sim-fm9001 rega rega)


(sim-fm9001 regb update)


(sim-fm9001 update update :mem-state mem-state
            :hyps (or (equal mem-state 0) (equal mem-state 1)))


(sim-fm9001 reada0 reada3 :dtack-wait #.(w_reada0->reada3)
            :disable (reada3$step1))


(sim-fm9001 readb0 readb3
            :mem-state mem-state
            :hyps (or (equal mem-state 0) (equal mem-state 1))
            :dtack-wait #.(w_readb0->readb3)
            :disable (readb3$step1))
```

```
(sim-fm9001 write0 write3
            :mem-state mem-state
            :hyps (or (equal mem-state 0) (equal mem-state 1))
            :dtack-wait #.(w_write0->write3)
            :disable (write3$step1))


(sim-fm9001 sefa0 sefa1)


(sim-fm9001 sefb0 sefb1)


(sim-fm9001 sefb1 sefb1)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    Execution time functions.
;;;
;;;  For each of the "chunks" above we define a function that states exactly
;;;  how many clock cycles are necessary to complete the instruction execution
;;;  cycle from the given starting state.  In these functions, CLOCK is the
;;;  memory delay oracle, and I-REG and FLAGS are the corresponding processor
;;;  registers.  The functions below are defined in terms of Common Lisp
;;;  functions appearing in "expand-fm9001.lisp".  The Common Lisp function
;;;  T_initial-state->final-state is an expression for the amount of time
;;;  necessary to execute the indicated section of of the state diagram.
;;;  The Common Lisp function CT_initial-state->final-state is an expression
;;;  for the memory delay oracle at the end of the sequence.  The Common Lisp
;;;  function TIMING-IF-TREE creates an "IF" tree of calls to timing functions
;;;  based on the branching decisions that appear in the state diagrams.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn t_fetch0 (clock i-reg flags)
  #.(t_fetch0->fetch0))


(defn t_sefb1 (clock i-reg flags)
  (let ((time #.(t_sefb1->sefb1)))
    (let ((clock #.(ct_sefb1->sefb1)))
      (plus time #.(timing-if-tree 'sefb1)))))


(defn t_sefb0 (clock i-reg flags)
  (let ((time #.(t_sefb0->sefb1)))
    (let ((clock #.(ct_sefb0->sefb1)))
      (plus time #.(timing-if-tree 'sefb1)))))
```

```
(defn t_sefa0 (clock i-reg flags)
  (let ((time #.(t_sefa0->sefa1)))
    (let ((clock #.(ct_sefa0->sefa1)))
      (plus time #.(timing-if-tree 'sefa1)))))


(defn t_write0 (clock i-reg flags)
  (let ((time #.(t_write0->write3)))
    (let ((clock  #.(ct_write0->write3)))
      (plus time #.(timing-if-tree 'write3)))))


(defn t_update (clock i-reg flags)
  (let ((time #.(t_update->update)))
    (let ((clock  #.(ct_update->update)))
      (plus time #.(timing-if-tree 'update)))))


(defn t_readb0 (clock i-reg flags)
  (let ((time #.(t_readb0->readb3)))
    (let ((clock  #.(ct_readb0->readb3)))
      (plus time #.(timing-if-tree 'readb3)))))


(defn t_reada0 (clock i-reg flags)
  (let ((time #.(t_reada0->reada3)))
    (let ((clock  #.(ct_reada0->reada3)))
      (plus time #.(timing-if-tree 'reada3)))))


(defn t_regb (clock i-reg flags)
  (let ((time #.(t_regb->update)))
    (let ((clock #.(ct_regb->update)))
      (plus time #.(timing-if-tree 'update)))))


(defn t_rega (clock i-reg flags)
  (let ((time #.(t_rega->rega)))
    (let ((clock #.(ct_rega->rega)))
      (plus time #.(timing-if-tree 'rega)))))


(defn t_fetch1 (clock i-reg flags)
  (let ((time #.(t_fetch1->decode)))
    (let ((clock #.(ct_fetch1->decode)))
      (plus time #.(timing-if-tree 'decode)))))
```

14.66 "proofs.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    PROOFS.EVENTS
;;;
;;;  This file contains events that lead up to the proof that the DUAL-EVAL
;;;  netlist model correctly implements the high-level specification.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    OPERATING-INPUTS-P
;;;
;;;  This predicate recognizes mid-level input streams that do not reset the
;;;  machine, and do not include hold requests.
;;;
;;;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn operating-inputs-p (inputs n)
  (if (zerop n)
      t
    (and (equal (reset--input (car inputs)) t)
         (equal (hold--input (car inputs)) t)
         (bvp (pc-reg-input (car inputs)))
         (equal (length (pc-reg-input (car inputs))) 4)
         (operating-inputs-p (cdr inputs) (sub1 n)))))


(disable operating-inputs-p)
```

```
(prove-lemma open-operating-inputs-p (rewrite)
  (and
   (implies
    (zerop n)
    (equal (operating-inputs-p inputs n)
           t))
   (implies
    (not (zerop n))
    (equal (operating-inputs-p inputs n)
           (and (equal (reset--input (car inputs)) t)
                (equal (hold--input (car inputs)) t)
                (bvp (pc-reg-input (car inputs)))
                (equal (length (pc-reg-input (car inputs))) 4)
                (operating-inputs-p (cdr inputs) (sub1 n))))))
  ;;Hint
  ((enable operating-inputs-p)))

;;;  This lemma relates our two predicates on the input streams.


(prove-lemma operating-inputs-p-implies-run-inputs-p (rewrite)
  (implies
   (operating-inputs-p inputs n)
   (run-inputs-p inputs n))
  ;;Hint
  ((enable run-inputs-p operating-inputs-p)))


(prove-lemma operating-inputs-p-plus (rewrite)
  (equal (operating-inputs-p inputs (plus n m))
         (and (operating-inputs-p inputs n)
              (operating-inputs-p (nthcdr n inputs) m)))
  ;;Hint
  ((enable operating-inputs-p nthcdr plus)))


(prove-lemma operating-inputs-p-1 (rewrite)
  (implies
   (and (operating-inputs-p inputs n)
        (not (zerop n)))
   (operating-inputs-p inputs 1))
  ;;Hint
  ((enable operating-inputs-p)))
```

```
(prove-lemma operating-inputs-p-implies-hold--input (rewrite)
  (implies
   (and (operating-inputs-p inputs n)
        (not (zerop n)))
   (equal (hold--input (car inputs))
          t))
  ;;Hint
  ((enable operating-inputs-p)))
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    MICROCYCLES state
;;;    TOTAL-MICROCYCLES state inputs n
;;;
;;;    Given a low-level machine state at the beginning of the instruction
;;;    execution cycle, MICROCYCLES computes the number of clock cycles
;;;    necessary to excute the instruction, given that the machine is
;;;    neither reset nor held.  MICROCYCLES assumes that the processor is in
;;;    state FETCH1.  Notice that in state FETCH1 the instruction resides in
;;;    memory.
;;;
;;;    TOTAL-MICROCYCLES computes the number of clock cycles needed to
;;;    execute N instructions, again assuming that the machine is neither
;;;    reset nor held.  Notice here that it is necessary to simulate each
;;;    instruction.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn microcycles (state)
  (let ((machine-state (car state))
        (mem-state (cadr state)))
    (let
      ((regs            (car machine-state))
       (flags           (cadr machine-state))
       (pc-reg          (caddddddddddr machine-state))
       (mem             (car mem-state))
       (mem-clock       (caddr mem-state)))
      (let
        ((instr (read-mem (read-mem pc-reg (car regs)) mem)))
        (t_fetch1 mem-clock instr flags)))))


(defn total-microcycles (state inputs n)
  (if (zerop n)
      0
    (let ((microcycles (microcycles state)))
      (let ((new-state (run-fm9001 state inputs microcycles)))
        (plus microcycles
              (total-microcycles
               new-state
               (nthcdr microcycles inputs)
               (sub1 n)))))))
```

```
(disable total-microcycles)


(prove-lemma open-total-microcycles (rewrite)
  (and
   (implies
    (zerop n)
    (equal (total-microcycles state inputs n)
           0))
   (implies
    (not (zerop n))
    (equal (total-microcycles state inputs n)
           (let ((microcycles (microcycles state)))
             (let ((new-state (run-fm9001 state inputs microcycles)))
               (plus microcycles
                     (total-microcycles
                      new-state
                      (nthcdr microcycles inputs)
                      (sub1 n))))))))
  ;;Hint
  ((enable total-microcycles)
   (disable microcycles)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    MACROCYCLE-INVARIANT state pc
;;;
;;;  This is an invariant about the state of the machine at the beginning of
;;;  the instruction execution cycle:  all of the internal registers are
;;;  Boolean and properly sized; the machine is in state FETCH1 (ready to
;;;  execute an instruction);  the ADDR-OUT register contains the program
;;;  counter;  there are no pending writes in the register file;  HOLD-
;;;  and RESET- are not asserted;  the PC-REG is equal to (and remains equal
;;;  to) the specified PC; and the memory is well-formed and in a quiescent
;;;  state.
;;;
;;;  This is an important invariant; one of the more important proofs that
;;;  follows states that if we begin in this state, and run for
;;;  (MICROCYCLES state), then we will reach a state recognized by this
;;;  invariant.  This is the basis for the inductive proof that the
;;;  low-level machine implements the behavioral specification.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defn macrocycle-invariant (state pc)
  (let ((machine-state (car state))
        (mem-state (cadr state)))
    (let
      ((regs             (car machine-state))
       (flags            (cadr machine-state))
       (a-reg            (caddr machine-state))
       (b-reg            (cadddr machine-state))
       (i-reg            (caddddr machine-state))
       (data-out         (cadddddr machine-state))
       (addr-out         (caddddddr machine-state))
       (last-reset-      (cadddddddr machine-state))
       (last-dtack-      (caddddddddr machine-state))
       (last-hold-       (cadddddddddr machine-state))
       (pc-reg           (caddddddddddr machine-state))
       (cntl-state       (cadddddddddddr machine-state))
       (mem              (car mem-state))
       (mem-cntl         (cadr mem-state))
       (mem-clock        (caddr mem-state))
       (mem-count        (cadddr mem-state))
       (mem-dtack        (caddddr mem-state))
       (mem-last-rw-     (cadddddr mem-state))
       (mem-last-address (caddddddr mem-state))
       (mem-last-data    (cadddddddr mem-state)))
      (let
        ((regs-regs    (car regs))
         (regs-we      (cadr regs))
         (regs-data    (caddr regs))
         (regs-address (cadddr regs)))
        (and
         (all-ramp-mem 4 regs-regs)
         (memory-okp 4 32 regs-regs)
         (equal regs-we f)
         (bvp regs-data) (equal (length regs-data) 32)
         (bvp regs-address) (equal (length regs-address) 4)
         (bvp flags) (equal (length flags) 4)
         (equal a-reg (read-regs pc-reg regs))
         (bvp b-reg) (equal (length b-reg) 32)
         (bvp i-reg) (equal (length i-reg) 32)
         (bvp data-out) (equal (length data-out) 32)
         (equal addr-out (read-regs pc-reg regs))
         (equal last-reset- t)
         (boolp last-dtack-)
         (equal last-hold- t)
         (bvp pc-reg) (equal (length pc-reg) 4)
         (equal pc-reg pc)
         (equal cntl-state (cv_fetch1 mem-last-rw- pc-reg i-reg flags pc-reg))
         (memory-okp 32 32 mem)
         (equal mem-cntl 0)
         (numberp mem-count)
         (boolp mem-last-rw-)
         (bvp mem-last-address) (equal (length mem-last-address) 32)
         (properp mem-last-data)
         (equal (length mem-last-data) 32))))))
```

```
(prove-lemma macrocycle-invariant==>chip-system-invariant$help ()
  (let
    ((state
      (list (list
             (list regs-regs regs-we regs-data regs-address)
             flags a-reg b-reg i-reg
             data-out addr-out
             last-reset- last-dtack- last-hold-
             pc-reg cntl-state)
            (list mem cntl clock count dtack
                  last-rw- last-address last-data))))
    (implies
     (macrocycle-invariant state pc)
     (chip-system-invariant state)))
  ((disable car-cdr-elim)))
```

```
(prove-lemma macrocycle-invariant==>chip-system-invariant (rewrite)
  (implies
   (and (fm9001-state-structure state)
        (macrocycle-invariant state pc))
   (chip-system-invariant state))
  ;;Hint
  ((disable chip-system-invariant macrocycle-invariant
            fm9001-state-structure)
   (use (macrocycle-invariant==>chip-system-invariant$help
         (regs-regs        (caaar state))
         (regs-we          (cadaar state))
         (regs-data        (caddaar state))
         (regs-address     (cadddaar state))
         (flags            (cadar state))
         (a-reg            (caddar state))
         (b-reg            (cadddar state))
         (i-reg            (caddddar state))
         (data-out         (cadddddar state))
         (addr-out         (caddddddar state))
         (last-reset-      (cadddddddar state))
         (last-dtack-      (caddddddddar state))
         (last-hold-       (cadddddddddar state))
         (pc-reg           (caddddddddddar state))
         (cntl-state       (cadddddddddddar state))
         (mem              (caadr state))
         (cntl             (cadadr state))
         (clock            (caddadr state))
         (count            (cadddadr state))
         (dtack            (caddddadr state))
         (last-rw-         (cadddddadr state))
         (last-address     (caddddddadr state))
         (last-data        (cadddddddadr state)))
        (fm9001-state-as-a-list))))

;;;  MACROCYCLE-INVARIANT* is introduced to delay opening up the function
;;;  until the low-level machine has been completely rewritten. This should
;;;  save a tremendous amount of time in the coming proof.


(defn macrocycle-invariant* (state pc)
  (macrocycle-invariant state pc))


(disable macrocycle-invariant*)


(prove-lemma macrocycle-invariant*=macrocycle-invariant (rewrite)
  (equal (macrocycle-invariant* (cons x y) pc)
         (macrocycle-invariant (cons x y) pc))
  ;;Hint
  ((enable macrocycle-invariant*)
   (disable macrocycle-invariant)))
```

```
(prove-lemma macrocycle-invariant-is-invariant$help ()
  (let
    ((state (list (list
                   (list regs-regs regs-we regs-data regs-address)
                   flags a-reg b-reg i-reg
                   data-out addr-out
                   last-reset- last-dtack- last-hold-
                   pc-reg cntl-state)
                  (list mem cntl clock count dtack
                        last-rw- last-address last-data))))
    (let ((microcycles (microcycles state)))
      (implies
       (and (macrocycle-invariant state pc)
            (operating-inputs-p inputs microcycles))
       (macrocycle-invariant* (run-fm9001 state inputs microcycles) pc))))
  ;;Hint
  ((enable expand-*-connectives open-nth t_fetch1 write-regs regfile-okp
           reg-direct->not-reg-indirect
           fm9001-step)
   (enable-theory fm9001-hardware-state-accessors)
   (disable-theory fm9001-step-theory)
   (disable plus *1*plus plus-add1
            make-tree *1*make-tree
            open-run-inputs-p open-run-inputs-p-add1
            open-operating-inputs-p)))
```

```
(prove-lemma macrocycle-invariant-is-invariant (rewrite)
  (implies
   (and (fm9001-state-structure state)
        (macrocycle-invariant state pc)
        (operating-inputs-p inputs (microcycles state)))
   (macrocycle-invariant (run-fm9001 state inputs (microcycles state)) pc))
  ;;Hint
  ((use (macrocycle-invariant-is-invariant$help
         (regs-regs        (caaar state))
         (regs-we          (cadaar state))
         (regs-data        (caddaar state))
         (regs-address     (cadddaar state))
         (flags            (cadar state))
         (a-reg            (caddar state))
         (b-reg            (cadddar state))
         (i-reg            (caddddar state))
         (data-out         (cadddddar state))
         (addr-out         (caddddddar state))
         (last-reset-      (cadddddddar state))
         (last-dtack-      (caddddddddar state))
         (last-hold-       (cadddddddddar state))
         (pc-reg           (caddddddddddar state))
         (cntl-state       (cadddddddddddar state))
         (mem              (caadr state))
         (cntl             (cadadr state))
         (clock            (caddadr state))
         (count            (cadddadr state))
         (dtack            (caddddadr state))
         (last-rw-         (cadddddadr state))
         (last-address     (caddddddadr state))
         (last-data        (cadddddddadr state))
         (instr (read-mem
                 (read-regs (caddddddddddar state)
                            (list
                             (caaar state) (cadaar state) (caddaar state)
                             (cadddaar state)))
                 (caadr state)))
         (microcycles (microcycles state)))
        (fm9001-state-as-a-list))
   (enable macrocycle-invariant*)
   (disable fm9001-state-structure macrocycle-invariant
            t_fetch1)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    MAP-UP
;;;
;;;  Maps a low-level state to a high-level state.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
(defn p-map-up (p-state)
  (list (car (regs p-state))
        (flags p-state)))


(defn mem-map-up (mem-state)
  (car mem-state))


(defn map-up (state)
  (let ((p-state   (car  state))
        (mem-state (cadr state)))
    (list
     (p-map-up p-state)
     (mem-map-up mem-state))))


(disable map-up)

;;;  This rather obvious fact is stated in a non-obvious way.  We
;;;  will be "mapping-up" a low-level simulation, i.e., a call of
;;;  RUN-FM9001.  If MAP-UP opens before RUN-FM9001 is completely rewritten
;;;  to a new state, i.e., a CONS, then we will be rewriting RUN-FM9001
;;;  3 times for each path through the state diagram.  By stating the lemma
;;;  this way, we wait until RUN-FM9001 is completely rewritten before
;;;  extracting the 3 interesting bits, thus saving massive amounts of work.
```

```
(prove-lemma open-map-up (rewrite)
  (let ((state (cons x y)))
    (equal (map-up state)
           (let ((p-state (car state))
                 (mem-state (cadr state)))
             (list
              (list (car (regs p-state))
                    (flags p-state))
              (car mem-state)))))
  ;;Hint
  ((enable map-up)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    MIDDLE=HIGH
;;;
;;;  This is the proof that the architecture implements the specification
;;;  for the execution of a single instruction.  The time abstraction
;;;  between the behavioral-level specification and the implementation is a
;;;  critical part of this proof; that is, the implementation requires a
;;;  number of clock cycles to execute a single instruction while the
;;;  behavioral-level specification executes just one instruction.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

;;;  FM9001-STEP* is introduced to delay opening up the high-level
;;;  specification until the low-level machine has been completely rewritten.
;;;  This saves a fair amount of time in the coming proof.


(defn fm9001-step* (state pc-reg)
  (fm9001-step state pc-reg))


(disable fm9001-step*)


(prove-lemma fm9001-step*-lemma (rewrite)
  (equal (equal (cons x y)
                (fm9001-step* state pc-reg))
         (equal (cons x y)
                (fm9001-step state pc-reg)))
  ;;Hint
  ((enable fm9001-step*)))
```

```
(prove-lemma middle=high$help ()
  (let
    ((state (list (list
                   (list regs-regs regs-we regs-data regs-address)
                   flags a-reg b-reg i-reg
                   data-out addr-out
                   last-reset- last-dtack- last-hold-
                   pc-reg cntl-state)
                  (list mem cntl clock count dtack
                        last-rw- last-address last-data))))
    (let ((microcycles (microcycles state)))
      (implies
       (and (macrocycle-invariant state pc-reg)
            (operating-inputs-p inputs microcycles))
       (equal (map-up (run-fm9001 state inputs microcycles))
              (fm9001-step* (list (list regs-regs flags) mem) pc-reg)))))
  ;;Hint
  ((enable expand-*-connectives open-nth t_fetch1 write-regs regfile-okp
           reg-direct->not-reg-indirect
           fm9001-step)
   (enable-theory fm9001-hardware-state-accessors)
   (disable-theory fm9001-step-theory)
   (disable plus *1*plus plus-add1
            make-tree *1*make-tree
            open-run-inputs-p open-run-inputs-p-add1
            open-operating-inputs-p)))


(prove-lemma macrocycle-invariant==>pc-reg ()
  (implies
   (macrocycle-invariant state pc-reg)
   (equal (pc-reg (car state)) pc-reg))
  ;;Hint
  ((enable pc-reg macrocycle-invariant open-nth)))
```

```
(prove-lemma middle=high (rewrite)
  (implies
   (and (fm9001-state-structure state)
        (macrocycle-invariant state pc)
        (operating-inputs-p inputs (microcycles state)))
   (equal (map-up (run-fm9001 state inputs (microcycles state)))
          (fm9001-step (map-up state) pc)))
  ;;Hint
  ((use (macrocycle-invariant==>pc-reg (pc-reg pc))
        (middle=high$help
         (regs-regs       (caaar state))
         (regs-we         (cadaar state))
         (regs-data       (caddaar state))
         (regs-address    (cadddaar state))
         (flags           (cadar state))

         (a-reg           (caddar state))
         (b-reg           (cadddar state))
         (i-reg           (caddddar state))
         (data-out        (cadddddar state))
         (addr-out        (caddddddar state))
         (last-reset-     (cadddddddar state))
         (last-dtack-     (caddddddddar state))
         (last-hold-      (cadddddddddar state))
         (pc-reg          (caddddddddddar state))
         (cntl-state      (cadddddddddddar state))
         (mem             (caadr state))
         (cntl            (cadadr state))
         (clock           (caddadr state))
         (count           (cadddadr state))
         (dtack           (caddddadr state))
         (last-rw-        (cadddddadr state))
         (last-address    (caddddddadr state))
         (last-data       (cadddddddadr state))
         (instr (read-mem
                 (read-regs (caddddddddddar state)
                            (list
                             (caaar state) (cadaar state) (caddaar state)
                             (cadddaar state)))
                 (caadr state)))
         (microcycles (microcycles state)))
        (fm9001-state-as-a-list))
   (disable fm9001-state-structure macrocycle-invariant
            t_fetch1)
   (enable map-up regs flags pc-reg open-nth fm9001-step*)
   (expand (microcycles state))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   Final Correctness Proofs
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;  The proof that the register-transfer specification implements the
```

```
;;;  behavioral specification.


(prove-lemma FM9001-interpreter-correct (rewrite)
  (let
    ((clock-cycles (total-microcycles state inputs instructions)))
    (implies
     (and (fm9001-state-structure state)
          (macrocycle-invariant state pc)
          (operating-inputs-p inputs clock-cycles))
     (equal (map-up
             (run-fm9001
              state
              inputs
              clock-cycles))
            (FM9001-interpreter
             (map-up state)
             pc
             instructions))))
  ;; Hint
  ((induct (total-microcycles state inputs instructions))
   (disable fm9001-state-structure microcycles macrocycle-invariant)))

;;;   The above, and CHIP-SYSTEM=RUN-FM9001 (see "chip.events"), yields this
;;;   lemma that the DUAL-EVAL netlist implements the FM9001 specification.


(prove-lemma chip-system=fm9001-interpreter (rewrite)
  (let
    ((rtl-inputs (map-up-inputs inputs)))
    (let
      ((clock-cycles (total-microcycles state rtl-inputs instructions)))
      (implies
       (and (chip-system& netlist)
            (fm9001-state-structure state)
            (macrocycle-invariant state pc)
            (chip-system-operating-inputs-p inputs clock-cycles)
            (operating-inputs-p rtl-inputs clock-cycles))
       (equal (map-up (simulate-dual-eval-2
                       'chip-system inputs state netlist clock-cycles))
              (fm9001-interpreter (map-up state) pc instructions)))))
  ;;Hint
  ((disable fm9001-state-structure macrocycle-invariant map-up
            chip-system-invariant)))

;;;   Note that when register 15 is the program counter, FM9001-INTERPRETER
;;;   is the same as FM9001.
```

```
(prove-lemma fm9001=fm9001-interpreter ()
  (equal (fm9001 state n)
         (fm9001-interpreter state (make-list 4 t) n))
  ((enable fm9001 fm9001-interpreter)
   (disable fm9001-step)))

;;;  MAP-DOWN-RELATION simply states that the register file, flags, and memory
;;;  of the low-level-state are equal to the corresponding things in the
;;;  high-level state.


(defn map-down-relation (high-level-state low-level-state)
  (let ((high-level-p-state   (car high-level-state))
        (high-level-mem-state (cadr high-level-state))
        (low-level-p-state   (car low-level-state))
        (low-level-mem-state (cadr low-level-state)))
    (and (equal (regs high-level-p-state)
                (car (regs low-level-p-state)))
         (equal (flags high-level-p-state)
                (flags low-level-p-state))
         (equal high-level-mem-state
                (car low-level-mem-state)))))


(disable map-down-relation)

;;;  A minimal hypothesis about the high-level-state.


(defn high-level-state-structure (state)
  (and (properp state)
       (properp (car state))
       (equal (length (car state)) 2)
       (equal (length state) 2)))


(disable high-level-state-structure)


(prove-lemma map-down-relation-lemma (rewrite)
  (implies
   (and (high-level-state-structure high-level-state)
        (map-down-relation high-level-state low-level-state))
   (equal (map-up low-level-state)
          high-level-state))
  ;;Hint
  ((enable map-down-relation high-level-state-structure map-up regs flags
           open-nth)))

;;;  These two lemmas mimic the 2 major results above, except that we use
;;;  MAP-DOWN-RELATION to relate the high- and low-level states rather than the
;;;  MAP-UP function.
```

```
(prove-lemma FM9001-interpreter-correct$map-down (rewrite)
  (implies
   (and (high-level-state-structure high-level-state)
        (map-down-relation high-level-state low-level-state)
        (fm9001-state-structure low-level-state)
        (macrocycle-invariant low-level-state pc)
        (operating-inputs-p
         inputs (total-microcycles low-level-state inputs n)))
   (equal (map-up
           (run-fm9001
            low-level-state
            inputs
            (total-microcycles low-level-state inputs n)))
          (FM9001-interpreter
           high-level-state
           pc
           n)))
  ;;Hint
  ((disable-theory t)
   (enable-theory ground-zero)
   (use (map-down-relation-lemma)
        (fm9001-interpreter-correct (state low-level-state)
                                    (instructions n)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    FM9001=CHIP-SYSTEM
;;;
;;;   The proofs that follow were originally what we thought of as the "final"
;;;   correctness results.  On further analysis, these really aren't very
;;;   satisfying because they presume that any high-level state can be mapped
;;;   down to a very specific low-level-state.  (You *could* do it with the
;;;   scan chain ...). All we can really guarantee about the machine is that
;;;   you can reset it, and then move forward at the end of the reset sequence.
;;;   We leave these events here for historical interest.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn fm9001-machine-statep (p-state)
  (let ((regs (car p-state))
        (flags (cadr p-state)))
    (and (properp p-state)
         (equal (length p-state) 2)
         (all-ramp-mem 4 regs)
         (memory-okp 4 32 regs)
         (bvp flags)
         (equal (length flags) 4))))
```

```
(defn fm9001-statep (state)
  (let ((p-state (car  state))
        (memory  (cadr state)))
    (and (properp state)
         (equal (length state) 2)
         (fm9001-machine-statep p-state)
         (memory-okp 32 32 memory))))


(defn p-map-down (p-state)
  (let ((regs  (car  p-state))
        (flags (cadr p-state)))
    (list
     (list regs f (make-list 32 f) (make-list 4 t))
     flags
     (read-mem (make-list 4 t) regs)    ; a-reg
     (make-list 32 f)                   ; b-reg
     (make-list 32 f)                   ; i-reg
     (make-list 32 f)                   ; data-out
     (read-mem (make-list 4 t) regs)    ; addr-out
     t                                  ; reset
     t                                  ; dtack
     t                                  ; hold
     (make-list 4 t)                    ; pc-reg
     (cv_fetch1 t                       ; cntl-state
                (list t t t t)
                (make-list 32 f)
                (list t f f f)
                (make-list 4 t)))))


(defn mem-map-down (memory)
  (list memory 0 0 0 t t (make-list 32 f) (make-list 32 (x))))


(defn map-down (state)
  (let ((p-state (car  state))
        (memory  (cadr state)))
    (list
     (p-map-down p-state)
     (mem-map-down memory))))


(prove-lemma map-up-inverts-map-down (rewrite)
  (implies (fm9001-statep state)
           (equal (map-up (map-down state)) state))
  ((enable regs flags nth)
   (disable make-list *1*make-list cv_fetch1 *1*cv_fetch1
            memory-okp *1*memory-okp all-ramp-mem)))
```

```
(prove-lemma fm9001-statep-implies-fm9001-state-structure (rewrite)
  (implies (fm9001-statep state)
           (fm9001-state-structure (map-down state)))
  ((disable make-list *1*make-list cv_fetch1 *1*cv_fetch1
            memory-okp *1*memory-okp all-ramp-mem)))


(prove-lemma fm9001-statep-implies-macrocycle-invariant-lemma1 (rewrite)
  (equal (equal (cv_fetch1 t
                           (list t t t t)
                           (make-list 32 f)
                           (list t f f f)
                           (make-list 4 t))
                (cv_fetch1 t
                           (make-list 4 t)
                           (make-list 32 f)
                           z
                           (make-list 4 t)))
         t)
  ((enable cv_fetch1 carry-in-help c-flag op-code
           v_fetch1 append mpg make-list)))


(prove-lemma fm9001-statep-implies-macrocycle-invariant (rewrite)
  (implies (fm9001-statep state)
           (macrocycle-invariant (map-down state) (make-list 4 t)))
  ((enable length-make-list properp-make-list bvp-make-list)
   (disable make-list *1*make-list carry-in-help cv_fetch1 read-mem
            memory-okp *1*memory-okp all-ramp-mem)))


(prove-lemma fm9001=chip-system-lemma1 (rewrite)
  (equal (nth 10 (car (map-down state))) (list t t t t))
  ((enable nth)))
```

```
(prove-lemma fm9001=chip-system (rewrite)
  (implies (and (chip-system& netlist)
                (fm9001-statep state)
                (chip-system-operating-inputs-p
                 inputs
                 (total-microcycles (map-down state)
                                    (map-up-inputs inputs)
                                    n))
                (operating-inputs-p
                 (map-up-inputs inputs)
                 (total-microcycles (map-down state)
                                    (map-up-inputs inputs)
                                    n)))
           (equal (fm9001 state n)
                  (map-up
                   (simulate-dual-eval-2
                    'chip-system inputs
                    (map-down state)
                    netlist
                    (total-microcycles (map-down state)
                                       (map-up-inputs inputs)
                                       n)))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable fm9001-statep-implies-fm9001-state-structure
           fm9001-statep-implies-macrocycle-invariant
           map-up-inverts-map-down)
   (use (fm9001=fm9001-interpreter)
        (chip-system=fm9001-interpreter (state (map-down state))
                                        (instructions n)
                                        (pc (make-list 4 t))))))
```

```
(defn no-holds-reset-or-test (i c)
  (and (chip-system-operating-inputs-p i c)
       (operating-inputs-p (map-up-inputs i) c)))

;;;  Here is an informal statement of the following lemma,
;;;  FM9001=CHIP-SYSTEM-SUMMARY, which is the chief result proved
;;;  about the FM9001.  Let H be the hardware netlist that we have
;;;  constructed for the FM9001.  The lemma states that for each
;;;  FM9001 user-visible state, S, and for each positive integer, N,
;;;  there exists a positive integer C such that the result of running
;;;  the user-model of the FM9001 (the function FM9001) N steps can
;;;  instead be obtained by simulating H and S under the DUAL-EVAL
;;;  semantics for C steps.  In precisely stating the theorem, we
;;;  arrange to supply the DUAL-EVAL semantics with a transform
;;;  (obtained with MAP-DOWN) of S, and afterwards we do a reverse
;;;  transformation (with MAP-UP) to obtain the final user-visible
;;;  state.  Futhermore, in making the statement precise we stipulate
;;;  that the external stimuli, I, to the chip do not enable a hold, a
;;;  reset, or a test input for the C clock cycles.  The fact that N
;;;  and C are different reflects the fact that a single FM9001
;;;  instruction takes several clock cycles to emulate at the
;;;  DUAL-EVAL netlist level.  The precise statement of this
;;;  correctness result is:


(prove-lemma fm9001=chip-system-summary nil
  (let ((h (chip-system$netlist))
        (c (total-microcycles (map-down s) (map-up-inputs i) n)))
  (implies
   (and (fm9001-statep s)
        (no-holds-reset-or-test i c))
   (equal (fm9001 s n)
          (map-up
           (simulate-dual-eval-2 'chip-system i (map-down s) h c)))))
  ;; Hint
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable no-holds-reset-or-test
           *1*chip-system$netlist *1*chip-system&)
   (use (fm9001=chip-system
         (state s)
         (netlist (chip-system$netlist))
         (inputs i)
         (n n)))))
```

14.67 "approx.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    APPROX.EVENTS
;;;
;;;    By Matt Kaufmann
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;; Table of Contents:
;;;
;;; 1. INITIALIZATION
;;; 2. APPROXIMATION NOTIONS
;;; 3. notion of MONOTONICITY-PROPERTY for a module
;;; 4. MONOTONICITY LEMMAS FOR BOOLEAN FUNCTIONS
;;; 5. MONOTONICITY LEMMAS FOR PRIMITIVE HARDWARE COMPONENTS
;;;    except the RAM
;;; 6. MONOTONICITY LEMMA for the RAM
;;; 7. MONOTONICITY FOR ALL PRIMITIVES
;;; 8. MONOTONICITY OF DUAL-EVAL
;;; 9. MONOTONICITY FOR SIMULATION

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                  INITIALIZATION                  ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  Note: This file requires macros defined in "monotonicity-macros.lisp",
;;;  which is loaded by "sysdef.lisp".

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;               APPROXIMATION NOTIONS              ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Here we define (b-approx a1 a2), which holds when a1 and a2 are
;; equal unless a1 is (X) (or anything other than T, F, or (Z),
;; actually).  Then we extend this "bit approximation" notion to
;; bit vectors (V-APPROX V1 V2) and states [including RAM states]
;; (S-APPROX S1 S2).


(defn b-knownp (x)
  (or (equal x t)
      (equal x f)
      (equal x (z))))
```

```
(prove-lemma b-knownp-compound-recognizer (rewrite)
  (equal (b-knownp x)
         (or (truep x)
             (falsep x)
             (zp x))))


(disable b-knownp)

(disable *1*b-knownp)


(defn b-approx (a1 a2)
  (if (b-knownp a1)
      (equal a1 a2)
    t))


(defn v-approx (v1 v2)
  ;; implies that v1 and v2 have the same length
  (if (listp v1)
      (and (listp v2)
           (b-approx (car v1) (car v2))
           (v-approx (cdr v1) (cdr v2)))
    ;; need the following so that approximation by a known
    ;; value implies equality
    (equal v1 v2)))


(defn v-knownp (x)
  (if (listp x)
      (and (b-knownp (car x))
           (v-knownp (cdr x)))
    t))


(prove-lemma v-knownp-implies-v-approx-is-equal (rewrite)
  ;; stated in the contrapositive for rewrite rule purposes
  (implies (and (not (equal x y))
                (v-knownp x))
           (not (v-approx x y))))

;; necessary so that s-approx doesn't blow its stack! -- or
;; at least, was at one point
```

```
(disable plus-add1)

;; We now consider approximation for states.  We demand that they have
;; the same type, in order that PAIRLIST acts the same "kind of way"
;; on each.  This appears to be important, or so I thought at one
;; time, in proving monotonicity in bizarre cases where we call
;; PAIRLIST on non-lists.

;; I want to be able to prove (s-approx x x), so that I can instantiate
;; lemmas roughly of the form
;; (implies (s-approx x y) (equal (foo x y) (bar x y)))
;; to obtain, say,
;; (equal (foo x x) (bar x x)).


(defn mem-width () 32)


(defn s-approx (s1 s2)
  (cond
   ((or (listp s1) (listp s2))
    (if (listp s1)
        (if (listp s2)
            (and (s-approx (car s1) (car s2))
                 (s-approx (cdr s1) (cdr s2)))
          f)
      f))
   ;; the empty list (aes 31-Mar-92)
   ((or (equal s1 nil) (equal s2 nil))
    (equal s1 s2))
   ((or (ramp s1) (ramp s2))
    (if (ramp s1)
        (if (ramp s2)
            (v-approx (ram-guts s1) (ram-guts s2))
          f)
      f))
   ((or (romp s1) (romp s2))
    (if (romp s1)
        (if (romp s2)
            (v-approx (rom-guts s1) (rom-guts s2))
          f)
      f))
   ((or (stubp s1) (stubp s2))
    (if (stubp s1)
        (if (stubp s2)
            (v-approx (stub-guts s1) (stub-guts s2))
          f)
      f))
   ;; In the final case, we view s1 and s2 as values in four-valued logic,
   ;; where any "wrong" value is simply viewed as X
   (t (b-approx s1 s2))))
```

```
(prove-lemma s-approx-x-x (rewrite)
  (s-approx x x))


(defn good-s (s)
  (cond
   ((listp s)
    (and (good-s (car s))
         (good-s (cdr s))))
   ((ramp s)
    (and (equal (length (ram-guts s)) (mem-width))
         (properp (ram-guts s))))
   ((romp s)
    (and (equal (length (rom-guts s)) (mem-width))
         (properp (rom-guts s))))
   ((stubp s)
    (and (equal (length (stub-guts s)) (mem-width))
         (properp (stub-guts s))))
   (t t)))


(defn s-knownp (s)
  (cond
   ((listp s)
    (and (s-knownp (car s))
         (s-knownp (cdr s))))
   ;; the empty list (aes 31-Mar-92)
   ((equal s nil)
    T)
   ((ramp s)
    (v-knownp (ram-guts s)))
   ((romp s)
    (v-knownp (rom-guts s)))
   ((stubp s)
    (v-knownp (stub-guts s)))
   ;; In the final case, we view s as a value in four-valued logic,
   ;; where any "wrong" value is simply viewed as X.
   (t (b-knownp s))))


(prove-lemma s-knownp-implies-s-approx-is-equal (rewrite)
  (implies (and (s-knownp x)
                (not (equal x y)))
           (not (s-approx x y))))

;; It seems OK to enable this again:

(enable plus-add1)
```

```
(defn v-approx-alist (alist1 alist2)
  ;; Checks that both alists have the same "strip-cars", and that
  ;; v-approx holds of the strip-cdrs (if that were defined).
  (if (listp alist1)
      (and (listp alist2)
           (equal (caar alist1) (caar alist2))
           (b-approx (cdar alist1) (cdar alist2))
           (v-approx-alist (cdr alist1) (cdr alist2)))
    (nlistp alist2)))


(defn s-approx-alist (alist1 alist2)
  ;; Checks that both alists have the same "strip-cars", and that
  ;; s-approx holds of the strip-cdrs (if that were defined).
  ;;    This is different from v-approx-alist, in that the strip-cdrs
  ;; is a list of states rather than a single vector (i.e. rather than
  ;; a list of bits).
  (if (listp alist1)
      (and (listp alist2)
           (equal (caar alist1) (caar alist2))
           (s-approx (cdar alist1) (cdar alist2))
           (s-approx-alist (cdr alist1) (cdr alist2)))
    (nlistp alist2)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;   notion of MONOTONICITY-PROPERTY for a module   ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defn monotonicity-property (flag fn netlist a1 a2 s1 s2)
  ;; The flag is analogous to the flag in DUAL-EVAL.
  (case flag
        (0 (implies (and (v-approx a1 a2)
                         (s-approx s1 s2))
                    (v-approx (dual-eval 0 fn a1 s1 netlist)
                              (dual-eval 0 fn a2 s2 netlist))))
        (1 (implies (and (v-approx-alist a1 a2)
                         (alistp a1)
                         (alistp a2)
                         (s-approx-alist s1 s2)
                         (alistp s1)
                         (alistp s2))
                    (v-approx-alist (dual-eval 1 fn a1 s1 netlist)
                                    (dual-eval 1 fn a2 s2 netlist))))
        (2 (implies (and (v-approx a1 a2)
                         (s-approx s1 s2))
                    (s-approx (dual-eval 2 fn a1 s1 netlist)
                              (dual-eval 2 fn a2 s2 netlist))))
        (3 (implies (and (v-approx-alist a1 a2)
                         (alistp a1)
                         (alistp a2)
                         (s-approx-alist s1 s2)
                         (alistp s1)
                         (alistp s2))
                    (s-approx-alist (dual-eval 3 fn a1 s1 netlist)
                                    (dual-eval 3 fn a2 s2 netlist))))
        (otherwise t)))


(disable monotonicity-property)

;; Now we prove a bunch of trivial lemmas in order to help
;; us to control the proof later.


(prove-lemma monotonicity-property-consequence-0 (rewrite)
  (implies (and (monotonicity-property 0 fn netlist a1 a2 s1 s2)
                (v-approx a1 a2)
                (s-approx s1 s2))
           (v-approx (dual-eval 0 fn a1 s1 netlist)
                     (dual-eval 0 fn a2 s2 netlist)))
  ((use (monotonicity-property (flag 0)))))
```

```
(prove-lemma monotonicity-property-consequence-1 (rewrite)
  (implies (and (monotonicity-property 1 fn netlist a1 a2 s1 s2)
                (v-approx-alist a1 a2)
                (alistp a1)
                (alistp a2)
                (s-approx-alist s1 s2)
                (alistp s1)
                (alistp s2))
           (v-approx-alist (dual-eval 1 fn a1 s1 netlist)
                           (dual-eval 1 fn a2 s2 netlist)))
  ((use (monotonicity-property (flag 1)))))


(prove-lemma monotonicity-property-consequence-2 (rewrite)
  (implies (and (monotonicity-property 2 fn netlist a1 a2 s1 s2)
                (v-approx a1 a2)
                (s-approx s1 s2))
           (s-approx (dual-eval 2 fn a1 s1 netlist)
                     (dual-eval 2 fn a2 s2 netlist)))
  ((use (monotonicity-property (flag 2)))))


(prove-lemma monotonicity-property-consequence-3 (rewrite)
  (implies (and (monotonicity-property 3 fn netlist a1 a2 s1 s2)
                (v-approx-alist a1 a2)
                (alistp a1)
                (alistp a2)
                (s-approx-alist s1 s2)
                (alistp s1)
                (alistp s2))
           (s-approx-alist (dual-eval 3 fn a1 s1 netlist)
                           (dual-eval 3 fn a2 s2 netlist)))
  ((use (monotonicity-property (flag 3)))))


(prove-lemma monotonicity-property-opener-0 (rewrite)
  ;; need iff below rather than equal in order for the proof to go through
  (iff (monotonicity-property 0 fn netlist a1 a2 s1 s2)
       (implies (and (v-approx a1 a2)
                     (s-approx s1 s2))
                (v-approx (dual-eval 0 fn a1 s1 netlist)
                          (dual-eval 0 fn a2 s2 netlist))))
  ((enable monotonicity-property)))
```

```
(prove-lemma monotonicity-property-opener-1 (rewrite)
  ;; need iff below rather than equal in order for the proof to go through
  (iff (monotonicity-property 1 fn netlist a1 a2 s1 s2)
       (implies (and (v-approx-alist a1 a2)
                     (alistp a1)
                     (alistp a2)
                     (s-approx-alist s1 s2)
                     (alistp s1)
                     (alistp s2))
                (v-approx-alist (dual-eval 1 fn a1 s1 netlist)
                                (dual-eval 1 fn a2 s2 netlist))))
  ((enable monotonicity-property)))


(prove-lemma monotonicity-property-opener-2 (rewrite)
  ;; need iff below rather than equal in order for the proof to go through
  (iff (monotonicity-property 2 fn netlist a1 a2 s1 s2)
       (implies (and (v-approx a1 a2)
                     (s-approx s1 s2))
                (s-approx (dual-eval 2 fn a1 s1 netlist)
                          (dual-eval 2 fn a2 s2 netlist))))
  ((enable monotonicity-property)))


(prove-lemma monotonicity-property-opener-3 (rewrite)
  ;; need iff below rather than equal in order for the proof to go through
  (iff (monotonicity-property 3 fn netlist a1 a2 s1 s2)
       (implies (and (v-approx-alist a1 a2)
                     (alistp a1)
                     (alistp a2)
                     (s-approx-alist s1 s2)
                     (alistp s1)
                     (alistp s2))
                (s-approx-alist (dual-eval 3 fn a1 s1 netlist)
                                (dual-eval 3 fn a2 s2 netlist))))
  ((enable monotonicity-property)))


(disable monotonicity-property-opener-0)

(disable monotonicity-property-opener-1)

(disable monotonicity-property-opener-2)
```

```
(disable monotonicity-property-opener-3)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;    MONOTONICITY LEMMAS FOR BOOLEAN FUNCTIONS     ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; See the file monotonicity-macros.lisp for the definition
;; and a sample expansion of the macro MONOTONICITY-LEMMAS.
;; The idea is to show that basic boolean functions are monotone
;; with respect to the partial order defined by B-APPROX, i.e.
;; (roughly stated) that changing an input from (X) can only
;; change the output if it was formerly (X).


(monotonicity-lemmas
 (f-buf f-and))


(monotonicity-lemmas
 (f-and3 f-and4)
 ((disable f-and b-approx)))


(monotonicity-lemmas (f-not) ((enable boolp)))


(monotonicity-lemmas
 (f-nand f-nand3 f-nand4 f-nand5 f-nand6 f-nand8)
 ((disable f-and f-not b-approx)))


(monotonicity-lemmas (f-or))


(monotonicity-lemmas
 (f-or3 f-or4 f-nor f-nor3 f-nor4 f-nor5 f-nor6 f-nor8)
 ((disable f-or f-not b-approx)))


(monotonicity-lemmas (f-xor f-equv) ((enable boolp)))


(monotonicity-lemmas
 (f-xor3 f-equv3)
 ((disable f-xor f-equv b-approx)))


(monotonicity-lemmas (f-if ft-buf ft-wire f-pullup))
```

```
(disable-all
 f-buf f-and f-and3 f-and4 f-not f-nand f-nand3
 f-nand4 f-nand5 f-nand6 f-nand8 f-or f-or3 f-or4
 f-nor f-nor3 f-nor4 f-nor5 f-nor6 f-nor8
 f-xor f-xor3 f-equv f-equv3 f-if ft-buf ft-wire f-pullup)

;; Do not remove the following -- although MONOTONICITY-LEMMAS does
;; not occur explicitly below, it does get used by the macro
;; PROVE-PRIMITIVE-MONOTONICITY.

(deftheory monotonicity-lemmas
  (f-buf-monotone f-and-monotone f-and3-monotone f-and4-monotone
         f-not-monotone
         f-nand-monotone f-nand3-monotone
         f-nand4-monotone f-nand5-monotone f-nand6-monotone
         f-nand8-monotone f-or-monotone f-or3-monotone f-or4-monotone
         f-nor-monotone f-nor3-monotone f-nor4-monotone
         f-nor5-monotone f-nor6-monotone f-nor8-monotone
         f-xor-monotone f-xor3-monotone f-equv-monotone f-equv3-monotone
         f-if-monotone ft-buf-monotone ft-wire-monotone f-pullup-monotone))


(disable b-approx)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;             MONOTONICITY LEMMAS FOR              ;;;;;
;;;;;          PRIMITIVE HARDWARE COMPONENTS           ;;;;;
;;;;;                except the RAM                    ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Next, let's prove that some primitive hardware components are
;; monotone, in the sense that when an input or "leaf" of the state is
;; changed from (X), they change their values or states only by
;; "filling in" for (X)s.

;; [Obscure comment that I no longer understand exactly:]  I'll adopt
;; an approach similar to Bishop/Warren's, except that I need to
;; consider arbitrary input argument lists rather than those of
;; exactly the right length (e.g.  4 for AO2$VALUE).  Actually, I
;; didn't realize that such lemmas already existed until starting to
;; define my versions.

;; First, a couple of lemmas.


(prove-lemma dual-eval-0-primp (rewrite)
  (implies (primp name)
           (equal (dual-eval 0 name args state netlist)
                  (dual-apply-value name args state)))
  ((enable dual-eval)))
```

```
(prove-lemma dual-eval-2-primp (rewrite)
  (implies (primp name)
           (equal (dual-eval 2 name args state netlist)
                  (dual-apply-state name args state)))
  ((enable dual-eval)))


(prove-lemma s-approx-implies-b-approx (rewrite)
  ;; geez, total functions are something
  (implies (s-approx x y)
           (b-approx x y))
  ((enable b-approx)))


(prove-lemma f-buf-type-set (rewrite)
  (or (truep (f-buf x))
      (falsep (f-buf x))
      (xp (f-buf x)))
  ((enable f-buf boolp)))


(prove-lemma fourp-f-buf (rewrite)
  (fourp (f-buf x)))


(prove-lemma fourp-f-if (rewrite)
  (fourp (f-if x y z))
  ((enable f-if)
   ;; @@##
   (disable f-if-rewrite)))
```

```
(prove-lemma fourp-implies-s-approx-is-b-approx (rewrite)
  (implies (and (fourp x)
                (fourp y)
                (b-approx x y))
           (equal (s-approx x y)
                  t)))

;;;   >(mapcar 'car common-lisp-primp-database)
;;;   (AO2 AO4 AO6 AO7 B-AND B-AND3 B-AND4 B-EQUV B-EQUV3 B-IF B-NAND
;;;                ; B1I deleted from primp-database (aes)
;;;        B-NAND3 B-NAND4 B-NAND5 B-NAND6 B-NAND8 B-NBUF B-NOR B-NOR3 B-NOR4
;;;        B-NOR5 B-NOR6 B-NOR8 B-NOT B-NOT-B4IP B-NOT-IVAP B-OR B-OR3 B-OR4
;;;        B-XOR B-XOR3 DEL2 DEL4 DEL10 PROCMON DP-RAM-16X32 FD1 FD1S FD1SP
;;;        FD1SLP ID LS-NAND3 LS-BUF10 MEM-32X32 T-BUF T-WIRE PULLUP
;;;        TTL-BIDIRECT TTL-CLK-INPUT TTL-INPUT TTL-OUTPUT
;;;        TTL-OUTPUT-PARAMETRIC TTL-OUTPUT-FAST TTL-TRI-OUTPUT
;;;        TTL-TRI-OUTPUT-FAST VDD VDD-PARAMETRIC VSS)
;;;
;;;   >


;; B1I was deleted from the following because it is no longer in the
;; primp-database. (aes)

(prove-primitive-monotonicity
 (AO2 AO4 AO6 AO7 B-AND B-AND3 B-AND4 B-EQUV B-EQUV3 B-IF B-NAND
      B-NAND3 B-NAND4 B-NAND5 B-NAND6 B-NAND8 B-NBUF B-NOR B-NOR3 B-NOR4
      B-NOR5 B-NOR6 B-NOR8 B-NOT B-NOT-B4IP B-NOT-IVAP B-OR B-OR3 B-OR4
      B-XOR B-XOR3 DEL2 DEL4 DEL10 PROCMON

      FD1 FD1S FD1SP
      FD1SLP ID
      ;; @@## LS-NAND3 LS-BUF10

      RAM-ENABLE-CIRCUIT T-BUF T-WIRE PULLUP
      TTL-BIDIRECT TTL-CLK-INPUT TTL-INPUT TTL-OUTPUT
      TTL-OUTPUT-PARAMETRIC TTL-OUTPUT-FAST TTL-TRI-OUTPUT
      TTL-TRI-OUTPUT-FAST VDD VDD-PARAMETRIC VSS))

; Notice that we do NOT have above:
; (prove-primitive-monotonicity (DP-RAM-16X32))
; (prove-primitive-monotonicity mem-32x32)


;; B1I-monotone was deleted from the following because B1I is no longer
;; in the primp-database. (aes)
```

```
(deftheory primitives-monotone
  (AO2-monotone
   AO4-monotone AO6-monotone AO7-monotone B-AND-monotone
   B-AND3-monotone B-AND4-monotone B-EQUV-monotone B-EQUV3-monotone
   B-IF-monotone B-NAND-monotone
   B-NAND3-monotone B-NAND4-monotone B-NAND5-monotone
   B-NAND6-monotone B-NAND8-monotone B-NBUF-monotone B-NOR-monotone
   B-NOR3-monotone B-NOR4-monotone
   B-NOR5-monotone B-NOR6-monotone B-NOR8-monotone B-NOT-monotone
   B-NOT-B4IP-monotone B-NOT-IVAP-monotone B-OR-monotone
   B-OR3-monotone B-OR4-monotone
   B-XOR-monotone B-XOR3-monotone DEL2-monotone
   DEL4-monotone DEL10-monotone PROCMON-monotone
   FD1-monotone FD1S-monotone FD1SP-monotone
   FD1SLP-monotone ID-monotone
   ;;; @@##  LS-NAND3-monotone LS-BUF10-monotone
   RAM-ENABLE-CIRCUIT-monotone
   T-BUF-monotone T-WIRE-monotone PULLUP-monotone
   TTL-BIDIRECT-monotone TTL-CLK-INPUT-monotone
   TTL-INPUT-monotone TTL-OUTPUT-monotone
   TTL-OUTPUT-PARAMETRIC-monotone TTL-OUTPUT-FAST-monotone
   TTL-TRI-OUTPUT-monotone
   TTL-TRI-OUTPUT-FAST-monotone VDD-monotone
   VDD-PARAMETRIC-monotone VSS-monotone))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;         MONOTONICITY LEMMA for the RAM           ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; This part of the proof seemed to take close to a couple of weeks!
;; All the rest put together was only about 3 days.  The goal is
;; to prove the equivalent of (prove-primitive-monotonicity (DP-RAM-16X32)).


(prove-lemma bvp-rev1 (rewrite)
  (implies (and (bvp x) (bvp acc))
           (bvp (rev1 x acc)))
  ((enable bvp rev1)))


(prove-lemma bvp-reverse (rewrite)
  (implies (bvp x)
           (bvp (reverse x)))
  ((enable reverse)))


(prove-lemma v-approx-bvp (rewrite)
  (implies (bvp x)
           (equal (v-approx x y)
                  (equal x y)))
  ((enable b-knownp b-approx)))
```

```
(defn read-mem1-monotone-induction (v-addr mem1 mem2)
  (cond ((or (stubp mem1) (stubp mem2) (nlistp v-addr)
             (nlistp mem1) (nlistp mem2))
         t)
        ((car v-addr)
         (read-mem1-monotone-induction (cdr v-addr) (cdr mem1) (cdr mem2)))
        (t (read-mem1-monotone-induction (cdr v-addr) (car mem1) (car mem2)))))


(prove-lemma read-mem1-monotone (rewrite)
  (implies (and (bvp v-addr)
                (s-approx mem1 mem2))
           (v-approx (read-mem1 v-addr mem1)
                     (read-mem1 v-addr mem2)))
  ((enable read-mem1)
   (induct (read-mem1-monotone-induction v-addr mem1 mem2))))


(prove-lemma bvp-implies-v-knownp (rewrite)
  (implies (bvp x)
           (v-knownp x))
  ((enable bvp v-knownp boolp)))


(prove-lemma v-approx-x-x (rewrite)
             (v-approx x x)
             ((enable b-approx)))


(prove-lemma v-approx-implies-b-approx-nth (rewrite)
  (implies (and (equal (nth n x) c)
                (not (equal (nth n y) c))
                (b-knownp c))
           (not (v-approx x y)))
  ((enable b-approx b-knownp nth)))


(enable *1*b-knownp)


(prove-lemma v-approx-implies-subranges-equal (rewrite)
  (implies (and (v-approx x y)
                (bvp (subrange x i j)))
           (equal (subrange x i j)
                  (subrange y i j)))
  ((enable subrange bvp boolp b-approx b-knownp)))


(prove-lemma v-approx-bvp-subrange (rewrite)
  (implies (and (v-approx a1 a2)
                (bvp (subrange a1 i j)))
           (bvp (subrange a2 i j)))
  ((enable subrange bvp boolp b-approx b-knownp)))
```

```
(prove-lemma v-approx-make-list-x (rewrite)
  (implies (and (equal (length y) bits)
                (properp y))
           (v-approx (make-list bits (x))
                     y))
  ((enable make-list b-approx b-knownp)
   (induct (nth bits y))))


(prove-lemma read-mem-monotone (rewrite)
  (implies (and (bvp v-addr)
                (s-approx mem1 mem2))
           (v-approx (read-mem v-addr mem1)
                     (read-mem v-addr mem2)))
  ((enable read-mem)))


(prove-lemma equal-length-read-mem1 (rewrite)
  (implies (s-approx s1 s2)
           (equal (equal (length (read-mem1 a s2))
                         (length (read-mem1 a s1)))
                  t))
  ((enable read-mem1)))


(prove-lemma equal-length-read-mem (rewrite)
  (implies (s-approx s1 s2)
           (equal (equal (length (read-mem a s2))
                         (length (read-mem a s1)))
                  t))
  ((enable read-mem)))


(prove-lemma s-approx-implies-properp-read-mem1 (rewrite)
  (implies (and (s-approx s1 s2)
                (properp (read-mem1 a s1)))
           (properp (read-mem1 a s2)))
  ((enable read-mem1)))


(prove-lemma s-approx-implies-properp-read-mem (rewrite)
  (implies (and (s-approx s1 s2)
                (properp (read-mem a s1)))
           (properp (read-mem a s2)))
  ((enable read-mem)))
```

```
(prove-lemma dual-port-ram-value-monotone (rewrite)
  (implies
   (and (numberp address-lines)         ; just to get rid of the need to show
        (s-approx s1 s2)                ; subrange is same on all zerops.
        (v-approx a1 a2))
   (v-approx (dual-port-ram-value bits address-lines a1 s1)
             (dual-port-ram-value bits address-lines a2 s2))))


(prove-lemma eval$-append (rewrite)
  (implies (and (litatom x)
                (member x lst))
           (equal (eval$ t x (append (pairlist lst y) z))
                  (eval$ t x (pairlist lst y))))
  ((induct (pairlist lst y))))


(prove-lemma eval$-pairlist-cons (rewrite)
  (implies (litatom x)
           (equal (eval$ t x (pairlist (cons c lst) y))
                  (if (equal x c)
                      (car y)
                    (eval$ t x (pairlist lst (cdr y)))))))


(prove-lemma eval$-append-2 (rewrite)
  (implies (and (litatom x)
                (not (member x lst)))
           (equal (eval$ t x (append (pairlist lst y) z))
                  (eval$ t x z)))
  ((induct (pairlist lst y))))


(prove-lemma dual-apply-value-dp-ram-16x32-lemma-1 (rewrite)
  (equal (dual-apply-value 'dp-ram-16x32 a s)
         (eval$ t
                (cdr (lookup-module 'results
                                    (cdr (primp 'dp-ram-16x32))))
                (append (pairlist (cdr (lookup-module
                                        'inputs
                                        (cdr (primp 'dp-ram-16x32))))
                                  a)
                        (pairstates (cdr (lookup-module
                                          'states
                                          (cdr (primp 'dp-ram-16x32))))
                                    s))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable dual-apply-value primp2 primp-lookup)))

;; Note:  ground-zero is NOT enabled in the following "opening up" lemma
;; (see monotonicity-macros.lisp for its statement), but IS enabled in the
;; lemma after that.  My recollection is that they couldn't be combined without
;; things blowing up.
```

```
(prove-dual-apply-value-or-state-dp-ram-16x32-lemma-2 value)
```

```
(prove-lemma dual-apply-value-dp-ram-16x32-lemma-3 (rewrite)
  (equal (dual-apply-value 'dp-ram-16x32 a s)
         (dual-port-ram-value
          32 4
          (list (car a)
                (cadr a)
                (caddr a)
                (cadddr a)
                (caddddr a)
                (cadddddr a)
                (caddddddr a)
                (cadddddddr a)
                (caddddddddr a)
                (cadddddddddr a)
                (caddddddddddr a)
                (cadddddddddddr a)
                (caddddddddddddr a)
                (cadddddddddddddr a)
                (caddddddddddddddr a)
                (cadddddddddddddddr a)
                (caddddddddddddddddr a)
                (cadddddddddddddddddr a)
                (caddddddddddddddddddr a)
                (cadddddddddddddddddddr a)
                (caddddddddddddddddddddr a)
                (cadddddddddddddddddddddr a)
                (caddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddddddddddddddddddr a))
          s))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable eval$-append eval$-append-2 eval$-pairlist-cons
           pairstates dual-apply-value-dp-ram-16x32-lemma-2)
   (disable pairlist append)))
```

```
(defn dual-port-ram-value-body (bits a-address b-address wen state)
  (if (or (not (bvp a-address))
          (and (not (equal wen t))
               (or (not (bvp b-address))
                   (equal a-address b-address))))
      (make-list bits (x))
    (let ((val (read-mem a-address state)))
      (if (and (properp val)
               (equal (length val) bits))
          val
        (make-list bits (x))))))


(prove-lemma dual-port-ram-value-is-dual-port-ram-value-body (rewrite)
  (let ((a-address (subrange args 0 (sub1 address-lines)))
        (b-address (subrange args address-lines
                             (sub1 (times 2 address-lines))))
        (wen (nth (times 2 address-lines) args)))
    (equal (dual-port-ram-value bits address-lines args state)
           (dual-port-ram-value-body bits a-address b-address wen state)))
  ((disable-theory t)
   (enable dual-port-ram-value dual-port-ram-value-body)))


(prove-lemma dual-apply-value-dp-ram-16x32 (rewrite)
  (equal (dual-apply-value 'dp-ram-16x32 a s)
         (dual-port-ram-value 32 4 a s))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable dual-apply-value-dp-ram-16x32-lemma-3
           dual-port-ram-value-is-dual-port-ram-value-body
           open-subrange open-nth)))


(prove-lemma dp-ram-16x32-monotone-value (rewrite)
             (monotonicity-property 0 'dp-ram-16x32 netlist a1 a2 s1 s2)
             ((disable-theory t)
              (enable-theory ground-zero)
              (enable *1*primp monotonicity-property
                      monotonicity-property-opener-0
                      dual-eval-0-primp dual-port-ram-value-monotone
                      dual-apply-value-dp-ram-16x32)))

;; This concludes the ``value'' half of the RAM's monotonicity lemma.
;; We move on to the ``state'' half.


(disable v-knownp)

(disable s-knownp-implies-s-approx-is-equal)

(disable bvp)

(disable fourp-implies-s-approx-is-b-approx)
```

```
(disable v-approx)

(disable s-knownp)
```

```
(prove-lemma s-approx-opener (rewrite)
  (and (implies (or (listp s1) (listp s2))
                (equal (s-approx s1 s2)
                       (if (listp s1)
                           (if (listp s2)
                               (and (s-approx (car s1) (car s2))
                                    (s-approx (cdr s1) (cdr s2)))
                             f)
                         f)))
       ;; add case for empty list (aes 31-Mar-92)
       (implies (or (equal s1 nil) (equal s2 nil))
                (equal (s-approx s1 s2)
                       (equal s1 s2)))
       (implies (or (ramp s1) (ramp s2))
                (equal (s-approx s1 s2)
                       (if (ramp s1)
                           (if (ramp s2)
                               (v-approx (ram-guts s1) (ram-guts s2))
                             f)
                         f)))
       (implies (or (romp s1) (romp s2))
                (equal (s-approx s1 s2)
                       (if (romp s1)
                           (if (romp s2)
                               (v-approx (rom-guts s1) (rom-guts s2))
                             f)
                         f)))
       (implies (or (stubp s1) (stubp s2))
                (equal (s-approx s1 s2)
                       (if (stubp s1)
                           (if (stubp s2)
                               (v-approx (stub-guts s1) (stub-guts s2))
                             f)
                         f)))
       (implies (not (or (listp s1) (listp s2)
                         ;; add case for empty list (aes 31-Mar-92)
                         (equal s1 nil) (equal s2 nil)
                         (ramp s1) (ramp s2)
                         (romp s1) (romp s2)
                         (stubp s1) (stubp s2)))
                ;; In the final case, we view s1 and s2 as values in
                ;; four-valued logic, where any "wrong" value is simply viewed
                ;; as (X).
                (equal (s-approx s1 s2)
                       (b-approx s1 s2))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (expand (s-approx s1 s2)
           (s-approx nil s2)
           (s-approx nil nil)
           (s-approx s1 nil))))
```

```
(disable s-approx)


(disable *1*mem-width)

(disable mem-width)


(prove-lemma v-approx-implies-nth-does-not-go-from-f-to-t (rewrite)
  (implies (and (not (nth n a1))
                (nth n a2))
           (not (v-approx a1 a2)))
  ((enable nth v-approx b-approx)))


(prove-lemma write-mem1-opener (rewrite)
  (and (implies (stubp mem)
                (equal (write-mem1 v-addr mem value) mem))
       (implies (nlistp v-addr)
                (equal (write-mem1 v-addr mem value)
                       (if (ramp mem) (ram value) mem)))
       (implies (and (listp v-addr) (nlistp mem))
                (equal (write-mem1 v-addr mem value)
                       mem))
       (implies (and (listp mem) (listp v-addr) (car v-addr))
                (equal (write-mem1 v-addr mem value)
                       (cons (car mem)
                             (write-mem1 (cdr v-addr)
                                         (cdr mem)
                                         value))))
       (implies (and (listp mem) (listp v-addr) (not (car v-addr)))
                (equal (write-mem1 v-addr mem value)
                       (cons (write-mem1 (cdr v-addr)
                                         (car mem)
                                         value)
                             (cdr mem)))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (expand (write-mem1 v-addr mem value))))


(defn write-mem1-monotone-induction (v-addr mem1 mem2)
  (cond ((or (stubp mem1) (stubp mem2))
         t)
        ((nlistp v-addr)
         t)
        ((or (nlistp mem1) (nlistp mem2))
         t)
        ((car v-addr)
         (write-mem1-monotone-induction
          (cdr v-addr) (cdr mem1) (cdr mem2)))
        (t (write-mem1-monotone-induction
            (cdr v-addr) (car mem1) (car mem2)))))
```

```
(prove-lemma write-mem1-monotone (rewrite)
  (implies (and (s-approx mem1 mem2)
                (v-approx data1 data2)
                (properp data1)
                (properp data2)
                (equal (length data1) (mem-width))
                (equal (length data2) (mem-width)))
           (s-approx (write-mem1 v-addr mem1 data1)
                     (write-mem1 v-addr mem2 data2)))
  ((induct (write-mem1-monotone-induction v-addr mem1 mem2))
   (expand (s-approx mem1 mem2))))


(prove-lemma write-mem-monotone (rewrite)
  (implies (and (s-approx mem1 mem2)
                (v-approx data1 data2)
                (properp data1)
                (properp data2)
                (equal (length data1) (mem-width))
                (equal (length data2) (mem-width)))
           (s-approx (write-mem v-addr mem1 data1)
                     (write-mem v-addr mem2 data2)))
  ((enable write-mem)))


(prove-lemma v-approx-length (rewrite)
  (implies (v-approx v1 v2)
           (not (lessp (length v1) (length v2))))
  ((enable v-approx length b-approx)))


(defn write-mem1-double-induction (v-addr mem1 mem2)
  (if (or (stubp mem1) (stubp mem2))
      t
    (if (nlistp v-addr)
        t
      (if (or (nlistp mem1) (nlistp mem2))
          t
        (if (car v-addr)
            (write-mem1-double-induction
             (cdr v-addr) (cdr mem1) (cdr mem2))
          (write-mem1-double-induction
           (cdr v-addr) (car mem1) (car mem2)))))))
```

```
(prove-lemma s-approx-write-mem1-id (rewrite)
  (implies (and (s-approx s1 s2)
                ;(good-s s1)
                (good-s s2)
                )
           (s-approx (write-mem1 v-addr
                                 s1
                                 (make-list (mem-width) (x)))
                     s2))
  ((induct (write-mem1-double-induction v-addr s1 s2))
   (expand (s-approx s1 s2))))


(prove-lemma s-approx-write-mem-id (rewrite)
  (implies (and (good-s s2)
                (s-approx s1 s2))
           (s-approx (write-mem v-addr
                                s1
                                (make-list (mem-width) (x)))
                     s2))
  ((enable write-mem)))


(prove-lemma s-approx-constant-ram-x-id (rewrite)
  (implies (and (good-s s2)
                (s-approx s1 s2))
           (s-approx (constant-ram s1
                                   (make-list (mem-width) (x)))
                     s2))
  ((enable constant-ram)
   (induct (s-approx s1 s2))))


(prove-lemma s-approx-constant-ram-x-constant-ram-x (rewrite)
  (implies (s-approx s1 s2)
           (s-approx (constant-ram s1
                                   (make-list (mem-width) (x)))
                     (constant-ram s2
                                   (make-list (mem-width) (x)))))
  ((enable constant-ram)
   (induct (s-approx s1 s2))))


(prove-lemma v-approx-preserves-length (rewrite)
  (implies (v-approx a1 a2)
           (equal (length a1) (length a2)))
  ((enable v-approx length)))
```

```
(prove-lemma v-approx-subrange (rewrite)
  (implies (v-approx a1 a2)
           (v-approx (subrange a1 i j)
                     (subrange a2 i j)))
  ((enable v-approx b-approx subrange)))


(prove-lemma mem-width-non-zero (rewrite)
  (not (equal (mem-width) 0))
  ((enable *1*mem-width)))

;; The following wouldn't be necessary if we could just enable
;; things at the goal level -- I only want to enable these two
;; (in the hint below) for case 3.


(prove-lemma s-approx-constant-ram-x-write-mem1-case-3 (rewrite)
  (implies (and (not (stubp s1))
                (not (stubp s2))
                (listp v-addr)
                (not (listp s1))
                (s-approx s1 s2)
                (properp v)
                (equal (length v) (mem-width))
                (not (ramp s1)))
           (s-approx s1
                     (write-mem1 v-addr s2 v)))
  ((enable write-mem1 s-approx)))


(prove-lemma s-approx-constant-ram-x-write-mem1 (rewrite)
  (implies (and (s-approx s1 s2)
                (properp v)
                (equal (length v) (mem-width))
                (good-s s2))
           (s-approx (constant-ram s1
                                   (make-list (mem-width) (x)))
                     (write-mem1 v-addr s2 v)))
  ((enable constant-ram)
   (induct (write-mem1-double-induction v-addr s1 s2))))


(prove-lemma s-approx-constant-ram-x-write-mem (rewrite)
  (implies (and (s-approx s1 s2)
                (properp v)
                (equal (length v) (mem-width))
                (good-s s2))
           (s-approx (constant-ram s1
                                   (make-list (mem-width) (x)))
                     (write-mem v-addr s2 v)))
  ((enable write-mem)))
```

```
(prove-lemma v-approx-preserves-properp (rewrite)
  (implies (not (iff (properp v1) (properp v2)))
           (not (v-approx v1 v2)))
  ((enable properp v-approx)))

;; Here's something silly, needed because mem-width is disabled below.

(prove-lemma mem-width-linear-facts (rewrite)
  (and (lessp 31 (mem-width))
       (lessp (mem-width) 33))
  ((enable *1*mem-width)))

;; A main goal!


(prove-lemma dual-port-ram-state-monotone nil
  (implies
   (and (s-approx s1 s2)
        (good-s s2)
        (v-approx a1 a2))
   (s-approx (dual-port-ram-state (mem-width) 4 a1 s1)
             (dual-port-ram-state (mem-width) 4 a2 s2))))


(prove-lemma dual-port-ram-state-monotone-rewrite (rewrite)
  (implies
   (and (s-approx s1 s2)
        (good-s s2)
        (v-approx a1 a2))
   (s-approx (dual-port-ram-state 32 4 a1 s1)
             (dual-port-ram-state 32 4 a2 s2)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable *1*mem-width)
   (use (dual-port-ram-state-monotone))))

;; The lemmas now leading up to DUAL-APPLY-STATE-DP-RAM-16X32 are patterned
;; (without much thought) after the corresponding ones for the value case.
;; Sort of, anyhow.
```

```
(prove-lemma dual-apply-state-dp-ram-16x32-lemma-1 (rewrite)
  (equal (dual-apply-state 'dp-ram-16x32 a s)
         (eval$ t
                (cdr (lookup-module 'new-states
                                    (cdr (primp 'dp-ram-16x32))))
                (append (pairlist (cdr (lookup-module
                                        'inputs
                                        (cdr (primp 'dp-ram-16x32))))
                                  a)
                        (pairstates (cdr (lookup-module
                                          'states
                                          (cdr (primp 'dp-ram-16x32))))
                                    s))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable dual-apply-state primp2 primp-lookup)))

;; Note:  ground-zero is NOT enabled in the following "opening up" lemma
;; (see monotonicity-macros.lisp for its statement), but IS enabled in the
;; lemma after that.  My recollection is that they couldn't be combined without
;; things blowing up.

(prove-dual-apply-value-or-state-dp-ram-16x32-lemma-2 state)
```

```
(prove-lemma dual-apply-state-dp-ram-16x32-lemma-3 (rewrite)
  (equal (dual-apply-state 'dp-ram-16x32 a s)
         (dual-port-ram-state
          32 4
          (list (car a)
                (cadr a)
                (caddr a)
                (cadddr a)
                (caddddr a)
                (cadddddr a)
                (caddddddr a)
                (cadddddddr a)
                (caddddddddr a)
                (cadddddddddr a)
                (caddddddddddr a)
                (cadddddddddddr a)
                (caddddddddddddr a)
                (cadddddddddddddr a)
                (caddddddddddddddr a)
                (cadddddddddddddddr a)
                (caddddddddddddddddr a)
                (cadddddddddddddddddr a)
                (caddddddddddddddddddr a)
                (cadddddddddddddddddddr a)
                (caddddddddddddddddddddr a)
                (cadddddddddddddddddddddr a)
                (caddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddddddddddddddddr a)
                (cadddddddddddddddddddddddddddddddddddddddr a)
                (caddddddddddddddddddddddddddddddddddddddddr a))
          s))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable eval$-append eval$-append-2 eval$-pairlist-cons
           pairstates dual-apply-state-dp-ram-16x32-lemma-2)
   (disable pairlist append)))
```

```
(defn dual-port-ram-state-body (bits b-address wen data state)
  (if (equal wen t)
      state
    ;;  There is a potential write.  If the address is unknown, wipe out the
    ;;  state.
    (if (not (bvp b-address))
        ;; omitting the things about the length and properness of args here
        (constant-ram state (make-list bits (x)))
      ;;  If WEN is solidly low, update the state with data, otherwise X out
      ;;  the addressed entry.
      (if (equal wen f)
          (write-mem b-address state data)
        (write-mem b-address state (make-list bits (x)))))))


(prove-lemma dual-port-ram-state-is-dual-port-ram-state-body (rewrite)
  (let ((b-address (subrange args address-lines
                             (sub1 (times 2 address-lines))))
        (wen (nth (times 2 address-lines) args))
        (data (subrange args
                        (add1 (times 2 address-lines))
                        (plus (times 2 address-lines) bits))))
    (equal (dual-port-ram-state bits address-lines args state)
           (dual-port-ram-state-body bits b-address wen data state)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable dual-port-ram-state dual-port-ram-state-body)))


(prove-lemma dual-apply-state-dp-ram-16x32 (rewrite)
  (equal (dual-apply-state 'dp-ram-16x32 a s)
         (dual-port-ram-state 32 4 a s))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable dual-apply-state-dp-ram-16x32-lemma-3
           dual-port-ram-state-is-dual-port-ram-state-body
           open-subrange open-nth)))

;;; Here's the state half of the RAM's monotonicity

(prove-lemma dp-ram-16x32-monotone-state (rewrite)
  (implies (good-s s2)
           (monotonicity-property 2 'dp-ram-16x32 netlist a1 a2 s1 s2))
  ((disable-theory t)
   (enable dual-port-ram-state-monotone-rewrite
           dual-apply-state-dp-ram-16x32
           dual-eval-2-primp
           monotonicity-property-opener-2
           and implies *1*primp)))
```

```
(prove-lemma dp-ram-16x32-monotone (rewrite)
  (and (monotonicity-property 0 'dp-ram-16x32 netlist a1 a2 s1 s2)
       (implies (good-s s2)
                (monotonicity-property 2 'dp-ram-16x32 netlist a1 a2 s1 s2))))

;; Originally the monotonicity lemma just above was an ADD-AXIOM from
;; which the rest of the proof followed.  So, we put the enabled/disabled
;; state back to where it was (at least approximately) before all these events.


(revert-state bvp-rev1)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;         MONOTONICITY FOR ALL PRIMITIVES          ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Here we recognize circuits that do not contain mem-32x32 (or
;; whatever EXCEPTIONS may be -- i.e. we'll apply it to '(mem-32x32)),
;; with the recognizer OK-NETLISTP.  Then we prove monotonicity for
;; all primitives except mem-32x32, which will be important in the
;; base step of our proof of monotonicty for all cases satisfying
;; OK-NETLISTP in the next section "MONOTONICITY OF DUAL-EVAL".
```

```
(defn ok-netlistp (flag fn netlist exceptions)
  ;; Here exceptions is a list of "bad" primitive names
  ;; that we are not allowed to use.  The flag is like
  ;; in dual-eval: if 0 or 2, then fn is a single module name, else
  ;; fn is a list of module occurrences.  Actually 0 and 2 behave the
  ;; same, as do 1 and 3 (see OK-NETLISTP-REDUCTION-REWRITE somewhat
  ;; below), but it seemed that the proofs were more likely to go
  ;; through easily if I used the same recursion structure here as is
  ;; used for dual-eval.
  (case flag
        (0 (if (primp fn)
               (not (member fn exceptions))
             (let ((module (lookup-module fn netlist)))
               (if (listp module)
                   (ok-netlistp 1
                                (module-occurrences module)
                                (delete-module fn netlist)
                                exceptions)
                 f))))
        (1 (let ((body fn))
             (if (listp body)
                 (let ((occurrence (car body)))
                   (let ((fn (occ-function occurrence)))
                     (and (ok-netlistp 0 fn netlist exceptions)
                          (ok-netlistp 1 (cdr body) netlist exceptions))))
               t)))
        (2 (if (primp fn)
               (not (member fn exceptions))
             (let ((module (lookup-module fn netlist)))
               (if (listp module)
                   (ok-netlistp 3
                                (module-occurrences module)
                                (delete-module fn netlist)
                                exceptions)
                 f))))
        (3 (let ((body fn))
             (if (listp body)
                 (let ((occurrence (car body)))
                   (let ((fn (occ-function occurrence)))
                     (and (ok-netlistp 2 fn netlist exceptions)
                          (ok-netlistp 3 (cdr body) netlist exceptions))))
               t)))
        (otherwise f))

  ((ord-lessp (cons (add1 (count netlist)) (count fn)))))
```

```
(prove-lemma primp-monotone (rewrite)
  (implies
   (and (primp fn)
        (or (equal flag 0)
            (and (equal flag 2)
                 (implies (equal fn 'dp-ram-16x32)
                          (good-s s2))))
        (not (equal fn 'mem-32x32)))
   (and (monotonicity-property flag fn netlist a1 a2 s1 s2)
        (monotonicity-property flag fn netlist a1 a2 s1 s2)))
  ((disable-theory t)
   (enable-theory ground-zero primitives-monotone)
   (enable primp lookup-module *1*primp-database
           dp-ram-16x32-monotone)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;            MONOTONICITY OF DUAL-EVAL             ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defn dual-eval-monotone-induction (flag fn netlist a1 a2 s1 s2)
  ;; straightforward adaptation of the scheme in dual-eval for
  ;; pairs of inputs and pairs of states
  (case flag
    (0 (if (primp fn)
           t
         (let ((module (lookup-module fn netlist)))
           (if (listp module)
               (let ((inputs  (module-inputs module))
                     (outputs (module-outputs module))
                     (occurrences (module-occurrences module))
                     (statenames (module-statenames module)))
                 (dual-eval-monotone-induction
                  1
                  occurrences
                  (delete-module fn netlist)
                  (pairlist inputs a1)
                  (pairlist inputs a2)
                  (pairstates statenames s1)
                  (pairstates statenames s2)))
             t))))
    (1 (let ((body fn)
             (occurrence (car fn)))
         (if (listp body)
             (let ((occ-name (occ-name occurrence))
                   (outputs (occ-outputs occurrence))
                   (fn (occ-function occurrence))
                   (inputs (occ-inputs occurrence)))
               (and
                (dual-eval-monotone-induction 0
                                              fn
                                              netlist
                                              (collect-value inputs a1)
                                              (collect-value inputs a2)
                                              (value occ-name s1)
                                              (value occ-name s2))
                (dual-eval-monotone-induction
                 1
                 (cdr body)
                 netlist
                 (append
                  (pairlist outputs
                            (dual-eval 0
                                       fn
                                       (collect-value inputs a1)
                                       (value occ-name s1)
                                       netlist))
                  a1)
                 (append
                  (pairlist outputs
                            (dual-eval 0
                                       fn
                                       (collect-value inputs a2)
                                       (value occ-name s2)
```

```
                                       netlist))
                 a2)
                s1 s2)))
          t)))
    (2 (if (primp fn)
          t
        (let ((module (lookup-module fn netlist)))
          (if (listp module)
              (let ((inputs  (module-inputs module))
                    (outputs (module-outputs module))
                    (occurrences (module-occurrences module))
                    (statenames (module-statenames module)))
                (and (dual-eval-monotone-induction
                      1
                      occurrences
                      (delete-module fn netlist)
                      (pairlist inputs a1)
                      (pairlist inputs a2)
                      (pairstates statenames s1)
                      (pairstates statenames s2))
                     (dual-eval-monotone-induction
                      3
                      occurrences
                      (delete-module fn netlist)
                      (dual-eval 1
                                 occurrences
                                 (pairlist inputs a1)
                                 (pairstates statenames s1)
                                 (delete-module fn netlist))
                      (dual-eval 1
                                 occurrences
                                 (pairlist inputs a2)
                                 (pairstates statenames s2)
                                 (delete-module fn netlist))
                      (pairstates statenames s1)
                      (pairstates statenames s2))))
            t))))
    (3 (let ((body fn))
         (if (listp body)
             (let ((occurrence (car body)))
               (let ((occ-name (occ-name occurrence))
                     (outputs (occ-outputs occurrence))
                     (fn (occ-function occurrence))
                     (inputs (occ-inputs occurrence)))
                 (and (dual-eval-monotone-induction
                       2
                       fn
                       netlist
                       (collect-value inputs a1)
                       (collect-value inputs a2)
                       (value occ-name s1)
                       (value occ-name s2))
                      (dual-eval-monotone-induction
                       3
                       (cdr body)
```

```
                       netlist
                       a1 a2 s1 s2))))
           t)))
    (otherwise t))
  ((ord-lessp (cons (add1 (count netlist)) (count fn)))))


(enable v-approx-x-x)


(prove-lemma v-approx-alist-implies-b-approx-value (rewrite)
  (implies (and (alistp alist1)
                (alistp alist2)
                (v-approx-alist alist1 alist2))
           (b-approx (value x alist1)
                     (value x alist2)))
  ((enable value alistp)))


(prove-lemma alistp-pairlist (rewrite)
  (alistp (pairlist x y))
  ((enable alistp pairlist)))


(prove-lemma alistp-append (rewrite)
  (implies (alistp x)
           (equal (alistp (append x y))
                  (alistp y)))
  ((enable alistp)))


(prove-lemma alistp-dual-eval-1 (rewrite)
  (implies (and (alistp bindings)
                ;; Silly statement is for induction heuristics,
                ;; though maybe that approach isn't necessary.
                (equal flag 1))
           (alistp (dual-eval flag
                              occurrences
                              bindings
                              state-bindints
                              netlist)))
  ((induct (dual-eval flag
                      occurrences
                      bindings
                      state-bindints
                      netlist))))
```

```
(prove-lemma v-approx-alist-implies-v-approx-collect-value (rewrite)
  (implies (and (alistp alist1)
                (alistp alist2)
                (v-approx-alist alist1 alist2))
           (v-approx (collect-value x alist1)
                     (collect-value x alist2)))
  ((enable collect-value)))


(defn s-approx-list (x y)
  (if (listp x)
      (and (listp y)
           (s-approx (car x) (car y))
           (s-approx-list (cdr x) (cdr y)))
    (nlistp y)))


(prove-lemma s-approx-list-implies-s-approx-alist (rewrite)
  (implies (s-approx-list s1 s2)
           (s-approx-alist (pairlist x s1)
                           (pairlist x s2)))
  ((enable pairlist)))


(prove-lemma v-approx-implies-v-approx-alist (rewrite)
  (implies (v-approx v1 v2)
           (v-approx-alist (pairlist x v1)
                           (pairlist x v2)))
  ((enable pairlist)))


(prove-lemma alistp-opener (rewrite)
  (equal (alistp (cons a z))
         (and (listp a)
              (alistp z)))
  ((enable alistp)))


(prove-lemma s-approx-alist-implies-s-approx-value (rewrite)
  (implies (and (s-approx-alist s1 s2)
                (alistp s1)
                (alistp s2))
           (s-approx (value w s1)
                     (value w s2)))
  ((enable value alistp)))


(prove-lemma v-approx-alist-append (rewrite)
  (implies (and (v-approx-alist a c)
                (v-approx-alist b d))
           (v-approx-alist (append a b) (append c d))))
```

```
(defn double-cdr-induction (x y)
  (if (and (listp x) (listp y))
      (double-cdr-induction (cdr x) (cdr y))
    t))


(prove-lemma s-approx-alist-implies-s-approx-list-collect-value (rewrite)
  (implies (and (s-approx-alist x y)
                (alistp x)
                (alistp y))
           (s-approx-list (collect-value a x) (collect-value a y)))
  ((enable collect-value value)))


(prove-lemma alistp-dual-eval-3 (rewrite)
  (implies (and (alistp state-bindings)
                ;; Silly statement is for induction heuristics,
                ;; though maybe that approach isn't necessary.
                (equal flag 3))
           (alistp (dual-eval flag
                              occurrences
                              bindings
                              state-bindings
                              netlist)))
  ((induct (dual-eval flag
                      occurrences
                      bindings
                      state-bindings
                      netlist))))


(prove-lemma s-approx-implies-s-approx-alist (rewrite)
  (implies (s-approx s1 s2)
           (s-approx-alist (pairlist x s1)
                           (pairlist x s2)))
  ((enable pairlist)))

;; The lemma above and the one below may either supersede,
;; subsume, or be subsumed by, corresponding lemmas about
;; s-approx-list -- or so I thought at some point.


(prove-lemma s-approx-alist-implies-s-approx-collect-value (rewrite)
  (implies (and (s-approx-alist x y)
                (alistp x)
                (alistp y))
           (s-approx (collect-value a x) (collect-value a y)))
  ((enable collect-value value)))
```

```
(prove-lemma ok-netlistp-reduction ()
  ;; The next lemma OK-NETLISTP-REDUCTION-REWRITE is the one
  ;; we actually want, but it's stated this way first for
  ;; the purpose of convenient proof by induction.
  (implies (or (equal flag 2) (equal flag 3))
           (equal (ok-netlistp flag fn netlist exceptions)
                  (if (equal flag 2)
                      (ok-netlistp 0 fn netlist exceptions)
                    (ok-netlistp 1 fn netlist exceptions)))))


(prove-lemma ok-netlistp-reduction-rewrite (rewrite)
  (and (equal (ok-netlistp 2 fn netlist exceptions)
              (ok-netlistp 0 fn netlist exceptions))
       (equal (ok-netlistp 3 fn netlist exceptions)
              (ok-netlistp 1 fn netlist exceptions)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (use (ok-netlistp-reduction (flag 2))
        (ok-netlistp-reduction (flag 3)))))


(prove-lemma dual-eval-monotone-no-ram (rewrite)
  (implies (ok-netlistp flag fn netlist '(mem-32x32 dp-ram-16x32))
           (monotonicity-property flag fn netlist a1 a2 s1 s2))
  ((induct (dual-eval-monotone-induction flag fn netlist a1 a2 s1 s2))
   (enable monotonicity-property)
   ;;; @@##
   (disable MEMBER-NON-LIST DUAL-EVAL-0-PRIMP)))


(defn good-s-alist (x)
  (if (listp x)
      (and (listp (car x))
           (good-s (cdar x))
           (good-s-alist (cdr x)))
    (equal x nil)))


(prove-lemma good-s-alist-pairlist (rewrite)
  (implies (good-s s)
           (good-s-alist (pairlist x s)))
  ((enable pairlist)))


(prove-lemma good-s-value (rewrite)
  (implies (good-s-alist x)
           (good-s (value w x)))
  ((enable value)))
```

```
(prove-lemma dual-eval-monotone (rewrite)
  (implies (ok-netlistp flag fn netlist '(mem-32x32))
           (implies (if (or (equal flag 0) (equal flag 2))
                        (good-s s2)
                      (good-s-alist s2))
                    (monotonicity-property flag fn netlist a1 a2 s1 s2)))
  ((induct (dual-eval-monotone-induction flag fn netlist a1 a2 s1 s2))
   (enable monotonicity-property)
   ;;; @@##
   (disable MEMBER-NON-LIST DUAL-EVAL-0-PRIMP)))


(prove-lemma good-s-collect-value (rewrite)
  (implies (good-s-alist s)
           (good-s (collect-value keys s)))
  ((enable collect-value value)))


(prove-lemma f-buf-preserves-good-s (rewrite)
  (good-s (f-buf x))
  ((enable f-buf boolp)))


(prove-lemma f-if-preserves-good-s (rewrite)
  (good-s (f-if x y z))
  ((enable f-if boolp)))


(prove-lemma good-s-0 (rewrite)
  (good-s 0)
  ((enable *1*good-s)))


(prove-primitive-preserves-good-s
 (AO2 AO4 AO6 AO7 B-AND B-AND3 B-AND4 B-EQUV B-EQUV3 B-IF B-NAND
      B-NAND3 B-NAND4 B-NAND5 B-NAND6 B-NAND8 B-NBUF B-NOR B-NOR3 B-NOR4
      B-NOR5 B-NOR6 B-NOR8 B-NOT B-NOT-B4IP B-NOT-IVAP B-OR B-OR3 B-OR4
      B-XOR B-XOR3 DEL2 DEL4 DEL10 PROCMON

      FD1 FD1S FD1SP
      FD1SLP ID
      ;; @@## LS-NAND3 LS-BUF10

      RAM-ENABLE-CIRCUIT T-BUF T-WIRE PULLUP
      TTL-BIDIRECT TTL-CLK-INPUT TTL-INPUT TTL-OUTPUT
      TTL-OUTPUT-PARAMETRIC TTL-OUTPUT-FAST TTL-TRI-OUTPUT
      TTL-TRI-OUTPUT-FAST VDD VDD-PARAMETRIC VSS))


(prove-lemma good-s-constant-ram (rewrite)
  (implies (good-s s)
           (good-s (constant-ram s (make-list 32 (x)))))
  ((enable constant-ram)))
```

```
(prove-lemma good-s-write-mem-1 (rewrite)
  (implies (and (good-s s)
                (properp value)
                (equal (length value) 32))
           (good-s (write-mem1 v-addr s value)))
  ((enable write-mem1)))


(prove-lemma good-s-write-mem (rewrite)
  (implies (and (good-s s)
                (properp value)
                (equal (length value) 32))
           (good-s (write-mem v-addr s value)))
  ((enable write-mem)))


(PROVE-LEMMA DP-RAM-16X32-PRESERVES-GOOD-S
             (REWRITE)
             (IMPLIES (GOOD-S S)
                      (GOOD-S (DUAL-APPLY-STATE 'DP-RAM-16X32
                                                ARGS S)))
             ((DISABLE-THEORY T)
              (ENABLE-THEORY GROUND-ZERO)
              (ENABLE *1*B-APPROX *1*V-APPROX *1*S-APPROX V-APPROX
                      good-s-constant-ram good-s-write-mem
                      length-subrange properp-subrange properp-make-list
                      length-make-list
                      DUAL-APPLY-STATE *1*PRIMP2 F-BUF-PRESERVES-GOOD-S
                      F-IF-PRESERVES-GOOD-S GOOD-S-0
                      dual-apply-state-dp-ram-16x32
                      DUAL-PORT-RAM-STATE
                      )))
```

```
(deftheory good-s-primitives-theory
  (AO2-preserves-good-s
   AO4-preserves-good-s AO6-preserves-good-s AO7-preserves-good-s
   B-AND-preserves-good-s
   B-AND3-preserves-good-s B-AND4-preserves-good-s B-EQUV-preserves-good-s
   B-EQUV3-preserves-good-s
   B-IF-preserves-good-s B-NAND-preserves-good-s
   B-NAND3-preserves-good-s B-NAND4-preserves-good-s B-NAND5-preserves-good-s
   B-NAND6-preserves-good-s B-NAND8-preserves-good-s B-NBUF-preserves-good-s
   B-NOR-preserves-good-s
   B-NOR3-preserves-good-s B-NOR4-preserves-good-s
   B-NOR5-preserves-good-s B-NOR6-preserves-good-s B-NOR8-preserves-good-s
   B-NOT-preserves-good-s
   B-NOT-B4IP-preserves-good-s B-NOT-IVAP-preserves-good-s
   B-OR-preserves-good-s
   B-OR3-preserves-good-s B-OR4-preserves-good-s
   B-XOR-preserves-good-s B-XOR3-preserves-good-s DEL2-preserves-good-s
   DEL4-preserves-good-s DEL10-preserves-good-s PROCMON-preserves-good-s
   FD1-preserves-good-s FD1S-preserves-good-s FD1SP-preserves-good-s
   FD1SLP-preserves-good-s ID-preserves-good-s
   ;;; @@##  LS-NAND3-preserves-good-s LS-BUF10-preserves-good-s
   RAM-ENABLE-CIRCUIT-preserves-good-s
   T-BUF-preserves-good-s T-WIRE-preserves-good-s PULLUP-preserves-good-s
   TTL-BIDIRECT-preserves-good-s TTL-CLK-INPUT-preserves-good-s
   TTL-INPUT-preserves-good-s TTL-OUTPUT-preserves-good-s
   TTL-OUTPUT-PARAMETRIC-preserves-good-s TTL-OUTPUT-FAST-preserves-good-s
   TTL-TRI-OUTPUT-preserves-good-s
   TTL-TRI-OUTPUT-FAST-preserves-good-s VDD-preserves-good-s
   VDD-PARAMETRIC-preserves-good-s VSS-preserves-good-s
   DP-RAM-16X32-PRESERVES-GOOD-S))


(prove-lemma primp-preserves-good-s (rewrite)
  (implies (and (good-s s)
                (primp fn)
                (not (equal fn 'mem-32x32)))
           (good-s (dual-apply-state fn args s)))
  ((disable-theory t)
   (enable-theory ground-zero good-s-primitives-theory)
   (enable primp lookup-module *1*primp-database)))
```

```
(prove-lemma good-s-preserved (rewrite)
  (implies (ok-netlistp flag fn netlist '(mem-32x32))
           (and (implies
                 (and (equal flag 2)
                      (good-s s))
                 (good-s (dual-eval flag fn a s netlist)))
                (implies
                 (and (equal flag 3)
                      (good-s-alist s))
                 (good-s-alist (dual-eval flag fn a s netlist)))))
  ((expand (dual-eval 2 fn a s netlist)
           (dual-eval 3 fn a s netlist))
   (induct (dual-eval flag fn a s netlist))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;           MONOTONICITY FOR SIMULATION            ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn v-approx-list (x y)
  ;; This is only to be used for the simulation lemma, so I imagine
  ;; that I don't care about the final cdrs.
  (if (listp x)
      (and (listp y)
           (v-approx (car x) (car y))
           (v-approx-list (cdr x) (cdr y)))
    (nlistp y)))


(defn v-s-approx-list (x y)
  ;; assumes x and y are lists of doublets
  (if (listp x)
      (and (listp y)
           (v-approx (caar x) (caar y))
           (s-approx (cadar x) (cadar y))
           (v-s-approx-list (cdr x) (cdr y)))
    t))


(defn nat-lst-lst-induction (n x y)
  (if (or (zerop n) (nlistp x) (nlistp y))
      t
    (nat-lst-lst-induction (sub1 n) (cdr x) (cdr y))))


(enable open-nth)
```

```
(prove-lemma v-approx-car-nth (rewrite)
  (implies (and (v-s-approx-list final-1 final-2)
                (lessp n (length final-1)))
           (v-approx (car (nth n final-1))
                     (car (nth n final-2))))
  ;; enabling nth wasn't enough to get the right induction
  ((induct (nat-lst-lst-induction n final-1 final-2))))


(prove-lemma s-approx-cadr-nth (rewrite)
  (implies (and (v-s-approx-list final-1 final-2)
                (lessp n (length final-1)))
           (s-approx (cadr (nth n final-1))
                     (cadr (nth n final-2))))
  ((induct (nat-lst-lst-induction n final-1 final-2))))

;; In the following lemma, I'd probably be happy if inputs-1
;; actually equals inputs-2.


(defn simulate-monotone-induction (fn inputs-1 state-1 netlist inputs-2
                                      state-2)
  (if (listp inputs-1)
      (simulate-monotone-induction
       fn
       (cdr inputs-1)
       (dual-eval 2 fn
                  (car inputs-1)
                  state-1 netlist)
       netlist
       (cdr inputs-2)
       (dual-eval 2 fn
                  (car inputs-2)
                  state-2 netlist))
    t))
```

```
(prove-lemma simulate-monotone (rewrite)
  (implies (and (v-approx-list inputs-1 inputs-2)
                (s-approx state-1 state-2)
                (good-s state-2)
                (ok-netlistp 0 fn netlist '(mem-32x32))
                (ok-netlistp 2 fn netlist '(mem-32x32)))
           (v-s-approx-list
            (simulate fn inputs-1 state-1 netlist)
            (simulate fn inputs-2 state-2 netlist)))
  ((induct
    (simulate-monotone-induction fn inputs-1 state-1 netlist inputs-2 state-2))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable simulate s-approx v-s-approx-list good-s-preserved
           monotonicity-property-consequence-0
           monotonicity-property-consequence-2 dual-eval-monotone
           v-approx-list)))


(prove-lemma v-approx-list-x-x (rewrite)
  (v-approx-list x x))


(defn doublet-p (x)
  (equal x (list (car x) (cadr x))))


(defn doublet-n-simulate-induction (fn inputs state netlist n)
  (if (zerop n)
      t
    (let ((new-state (dual-eval 2 fn (car inputs) state netlist)))
      (doublet-n-simulate-induction
       fn (cdr inputs) new-state netlist (sub1 n)))))


(prove-lemma doublet-n-simulate (rewrite)
  (implies (lessp n (length inputs))
           (doublet-p (nth n (simulate fn inputs state netlist))))
  ((induct (doublet-n-simulate-induction fn inputs state netlist n))
   (expand (simulate fn inputs state netlist))))


(prove-lemma doublet-p-equal-approx nil
  (implies (and (doublet-p x)
                (doublet-p y)
                (v-knownp (car x))
                (s-knownp (cadr x))
                (v-approx (car x) (car y))
                (s-approx (cadr x) (cadr y)))
           (equal x y)))


(disable doublet-p)
```

```
(prove-lemma length-simulate (rewrite)
  (equal (length (simulate fn inputs state netlist))
         (length inputs)))


(prove-lemma xs-suffice-for-reset-lemma-verbose nil
  (let ((final-1 (nth n (simulate fn inputs state-1 netlist)))
        (final-2 (nth n (simulate fn inputs state-2 netlist))))
    (implies (and (lessp n (length inputs))
                  (s-approx state-1 state-2)
                  (v-knownp (car final-1))
                  (s-knownp (cadr final-1))
                  (good-s state-2)
                  (ok-netlistp 0 fn netlist '(mem-32x32))
                  (ok-netlistp 2 fn netlist '(mem-32x32)))
             (equal final-1 final-2)))
  ((use (doublet-p-equal-approx
         (x (nth n (simulate fn inputs state-1 netlist)))
         (y (nth n (simulate fn inputs state-2 netlist)))))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable v-approx-car-nth s-approx-cadr-nth simulate-monotone
           v-approx-list-x-x doublet-n-simulate doublet-p-equal-approx
           length-simulate)))


(prove-lemma xs-suffice-for-reset-lemma nil
  (let ((final-1 (nth n (simulate fn inputs state-1 netlist)))
        (final-2 (nth n (simulate fn inputs state-2 netlist))))
    (implies (and (lessp n (length inputs))
                  (s-approx state-1 state-2)
                  (v-knownp (car final-1))
                  (s-knownp (cadr final-1))
                  (good-s state-2)
                  (ok-netlistp 0 fn netlist '(mem-32x32)))
             (equal final-1 final-2)))
  ((use (xs-suffice-for-reset-lemma-verbose))))
```

14.68 "final-reset.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    FINAL-RESET.EVENTS
;;;
;;;  This file contains the verification of the FM9001 reset sequence.  There
;;;  are several reset proofs in this file, all stated in slightly different
;;;  ways.  There are also a number of refinements of the main correctness
;;;  theorem of the FM9001 microprocessor.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;   Since some of the following proofs are simply simulations, we compile the
;;;   uncompiled DEFNs at this point to speed up these proofs.


(compile-uncompiled-defns* "a-weird-file-name-0")

;;;  The definition of a properly structured but completely unknown machine
;;;  state.


(defn unknown-regfile ()
  (list
   ;; regs
   (cons (cons (cons (cons (ram (make-list 32 (x)))
                           (ram (make-list 32 (x))))
                     (cons (ram (make-list 32 (x)))
                           (ram (make-list 32 (x)))))
               (cons (cons (ram (make-list 32 (x)))
                           (ram (make-list 32 (x))))
                     (cons (ram (make-list 32 (x)))
                           (ram (make-list 32 (x))))))
         (cons (cons (cons (ram (make-list 32 (x)))
                           (ram (make-list 32 (x))))
                     (cons (ram (make-list 32 (x)))
                           (ram (make-list 32 (x)))))
               (cons (cons (ram (make-list 32 (x)))
                           (ram (make-list 32 (x))))
                     (cons (ram (make-list 32 (x)))
                           (ram (make-list 32 (x)))))))
   (x)                                  ; we
   (make-list 32 (x))                   ; data
   (make-list 4 (x))                    ; address
   ))
```

```
(defn unknown-machine-state ()
  (list
   (unknown-regfile)                  ; regs
   (make-list 4 (x))                  ; flags
   (make-list 32 (x))                 ; a-reg
   (make-list 32 (x))                 ; b-reg
   (make-list 32 (x))                 ; i-reg
   (make-list 32 (x))                 ; data-out
   (make-list 32 (x))                 ; addr-out
   (x)                                ; reset
   (x)                                ; dtack
   (x)                                ; hold
   (make-list 4 (x))                  ; pc-reg
   (make-list 40 (x))                 ; cntl-state
   ))


(defn unknown-memory-state ()
  (list
   (stub (make-list 32 f))            ; mem
   0                                  ; cntl
   0                                  ; clock
   0                                  ; count
   (x)                                ; dtack-asserted
   (x)                                ; last-rw-
   (make-list 32 (x))                 ; last-address
   (make-list 32 (x))))               ; last-data


(defn unknown-state ()
  (list (unknown-machine-state) (unknown-memory-state)))


(prove-lemma chip-system-invariant-unknown-state ()
  (chip-system-invariant (unknown-state)))


(prove-lemma fm9001-state-structure-unknown-state ()
  (fm9001-state-structure (unknown-state)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    Resetting the CHIP-SYSTEM
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;  Putting together an input stream that initializes CHIP-SYSTEM.
```

```
(defn reset-vector ()
 (list* (x)                          ; clk
        (x)                          ; ti
        f                            ; te
        f                            ; reset-
        t                            ; hold-
        t                            ; disable-regfile-
        t                            ; test-regfile-
        (make-list 4 t)))            ; pc-reg


(defn run-vector ()
 (list* (x)                          ; clk
        (x)                          ; ti
        f                            ; te
        t                            ; reset-
        t                            ; hold-
        t                            ; disable-regfile-
        t                            ; test-regfile-
        (make-list 4 t)))            ; pc-reg


(defn reset-sequence ()
  (cons (reset-vector) (make-list 19 (run-vector))))


(prove-lemma chip-system-operating-inputs-p-reset-sequence ()
  (chip-system-operating-inputs-p (reset-sequence) 20))

;;;  The initialized state.
```

```
(defn initialized-regfile ()
  (list
   ;; regs
   (cons (cons (cons (cons (ram (make-list 32 f))
                           (ram (make-list 32 f)))
                     (cons (ram (make-list 32 f))
                           (ram (make-list 32 f))))
               (cons (cons (ram (make-list 32 f))
                           (ram (make-list 32 f)))
                     (cons (ram (make-list 32 f))
                           (ram (make-list 32 f)))))
         (cons (cons (cons (ram (make-list 32 f))
                           (ram (make-list 32 f)))
                     (cons (ram (make-list 32 f))
                           (ram (make-list 32 f))))
               (cons (cons (ram (make-list 32 f))
                           (ram (make-list 32 f)))
                     (cons (ram (make-list 32 f))
                           (ram (make-list 32 f))))))
   f                                  ; we
   (make-list 32 f)                   ; data
   (make-list 4 t)                    ; address
   ))


(defn initialized-machine-state ()
  (list
   (initialized-regfile)              ; regs
   (list t f f f)                     ; flags
   (make-list 32 f)                   ; a-reg
   (make-list 32 f)                   ; b-reg
   (make-list 32 f)                   ; i-reg
   (make-list 32 f)                   ; data-out
   (make-list 32 f)                   ; addr-out
   t                                  ; reset
   t                                  ; dtack
   t                                  ; hold
   (make-list 4 t)                    ; pc-reg
   (cv_fetch1                         ; cntl-state
              t                       ;   RW-
              (list t t t t)          ;   REGS-ADDRESS
              (make-list 32 f)        ;   I-REG
              (list t f f f)          ;   FLAGS
              (make-list 4 t))))      ;   PC-REG
```

```
(defn initialized-memory-state ()
  (list (stub (make-list 32 f))         ; mem
        0                               ; cntl
        0                               ; clock
        0                               ; count
        t                               ; dtack-asserted
        t                               ; last-rw-
        (make-list 32 f)                ; last-address
        (make-list 32 (x))              ; last-data
        ))


(defn final-state ()
  (list (initialized-machine-state) (initialized-memory-state)))

;;;   Prove that the reset sequence works, by running the machine.


(prove-lemma reset-works ()
  (equal (run-fm9001 (unknown-state)
                     (map-up-inputs (reset-sequence))
                     (length (reset-sequence)))
         (final-state))
  ;;Hint
  ((disable run-fm9001 unknown-state map-up-inputs reset-sequence
            length final-state)))


(prove-lemma unknown-state-okp ()
  (and
   (fm9001-state-structure (unknown-state))
   (chip-system-invariant (unknown-state))
   (chip-system-operating-inputs-p (reset-sequence) 20))
  ;;Hint
  ((disable chip-system-invariant
            chip-system-operating-inputs-p
            reset-sequence unknown-state
            fm9001-state-structure)))


(prove-lemma final-state-okp (rewrite)
  (and
   (chip-system& (chip-system$netlist))
   (fm9001-state-structure (final-state))
   (macrocycle-invariant (final-state) (make-list 4 t)))
  ;;Hint
  ((disable chip-system& chip-system$netlist
            fm9001-state-structure final-state
            macrocycle-invariant)))
```

```
(prove-lemma instance-theorem ()
  (let
    ((fm9001-inputs (map-up-inputs inputs))
     (netlist (chip-system$netlist))
     (state (final-state)))
    (let
      ((time (total-microcycles state fm9001-inputs n)))
      (implies
       (and (chip-system& netlist)
            (fm9001-state-structure state)
            (macrocycle-invariant state pc)
            (chip-system-operating-inputs-p inputs time)
            (operating-inputs-p fm9001-inputs time))
       (equal (map-up (simulate-dual-eval-2
                       'chip-system inputs state netlist time))
              (fm9001-interpreter (map-up state)
                                  pc n)))))
  ;; Hint
  ((disable map-up-inputs total-microcycles
            chip-system$netlist final-state *1*final-state
            fm9001-state-structure macrocycle-invariant
            chip-system-operating-inputs-p
            operating-inputs-p map-up simulate-dual-eval-2
            fm9001-interpreter pc-reg)))


(prove-lemma fm9001-statep-map-up-final-state (rewrite)
  (fm9001-statep (map-up (final-state)))
  ((disable fm9001-statep map-up final-state)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    Resetting the CHIP in isolation
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defn reset-vector-chip ()
  (list* (x)                           ; clk
         (x)                           ; ti
         f                             ; te

         (x)                           ; dtack-
         f                             ; reset-
         t                             ; hold-
         t                             ; disable-regfile-
         t                             ; test-regfile-

         (append
          (make-list 4 t)              ; pc-reg
          (make-list 32 (x)))))        ; data input


;;;  Note: If clk is (x), chip output PO will be (x).
;;;  First run vector (clk T).  PO will be F.


(defn run-vector-chip-1 ()
  (list* t                             ; clk
         (x)                           ; ti
         f                             ; te

         t                             ; dtack-
         t                             ; reset-
         t                             ; hold-
         t                             ; disable-regfile-
         t                             ; test-regfile-

         (append
          (make-list  4 t)             ; pc-reg
          (make-list 32 t))))          ; data input

;;;  Second run vector (clk F).  PO will be F.


(defn run-vector-chip-2 ()
  (list* f                             ; clk
         (x)                           ; ti
         f                             ; te

         t                             ; dtack-
         t                             ; reset-
         t                             ; hold-
         t                             ; disable-regfile-
         t                             ; test-regfile-

         (append
          (make-list  4 t)             ; pc-reg
          (make-list 32 t))))          ; data input
```

```
(defn reset-sequence-chip-1 ()
  (cons (reset-vector-chip) (make-list 19 (run-vector-chip-1))))


(defn reset-sequence-chip-2 ()
  (cons (reset-vector-chip) (make-list 19 (run-vector-chip-2))))


(prove-lemma reset-sequence-chip-1-vs-2 ()
  (let ((inputs-1 (reset-sequence-chip-1))
        (inputs-2 (reset-sequence-chip-2))
        (state    (unknown-machine-state))
        (netlist  (chip$netlist)))

    (equal (simulate 'chip inputs-1 state netlist)
           (simulate 'chip inputs-2 state netlist)))

  ;; hints
  ((disable reset-sequence-chip-1 reset-sequence-chip-2
            unknown-machine-state chip$netlist simulate)))


(prove-lemma simulate-reset-chip-final-state ()
  (let ((fn      'chip)
        (inputs  (reset-sequence-chip-1))
        (state   (unknown-machine-state))
        (netlist (chip$netlist)))

    (let ((n (sub1 (length inputs))))

      (let ((result (nth n (simulate fn inputs state netlist))))

        (let ((final-simulated-state (cadr result)))

          (equal final-simulated-state
                 (car (final-state)))))))

  ;; hints
  ((disable reset-sequence-chip-1 unknown-machine-state chip$netlist sub1
            length nth simulate final-state)))


;;;  This lemma says that the reset sequence is "good" in
;;;  the sense required by the last lemma below.
```

```
(prove-lemma for-final-1-of-reset-sequence-chip (rewrite)
  (let ((fn      'chip)
        (inputs  (reset-sequence-chip-1))
        (state-1 (unknown-machine-state))
        (netlist (chip$netlist)))

    (let ((n (sub1 (length inputs))))

      (let ((final-1 (nth n (simulate fn inputs state-1 netlist))))

        (and (equal (lessp n (length inputs)) t)
             (v-knownp (car final-1))
             (s-knownp (cadr final-1))
             (ok-netlistp 0 fn netlist '(mem-32x32))))))

  ;; hints
  ((disable reset-sequence-chip-1 unknown-machine-state chip$netlist sub1
            length nth simulate v-knownp s-knownp ok-netlistp)))


(disable for-final-1-of-reset-sequence-chip)

;;;  Thus, for any STATE-2, we can reset the machine so long
;;;  as (S-APPROX STATE-1 STATE-2) is true.  We believe that
;;;  (S-APPROX STATE-1 STATE-2) is true for any STATE-2 of the
;;;  proper form.
```

```
(prove-lemma xs-suffice-for-reset-chip-lemma-instance ()
  (let ((fn      'chip)
        (inputs  (reset-sequence-chip-1))
        (state-1 (unknown-machine-state))
        (netlist (chip$netlist)))

    (let ((n (sub1 (length inputs))))

      (let ((final-1 (nth n (simulate fn inputs state-1 netlist)))
            (final-2 (nth n (simulate fn inputs state-2 netlist))))

        (implies (and (s-approx state-1 state-2)
                      (good-s state-2))
                 (equal final-1 final-2)))))

  ;; hints
  ((disable reset-sequence-chip-1 *1*reset-sequence-chip-1
            unknown-machine-state *1*unknown-machine-state
            chip$netlist *1*chip$netlist
            sub1 *1*sub1 length *1*length nth *1*nth
            simulate *1*simulate s-approx *1*s-approx
            v-knownp *1*v-knownp s-knownp *1*s-knownp
            ok-netlistp *1*ok-netlistp)
   (enable for-final-1-of-reset-sequence-chip)
   (use (xs-suffice-for-reset-lemma
         (n       (sub1 (length (reset-sequence-chip-1))))
         (fn      'chip)
         (inputs  (reset-sequence-chip-1))
         (state-1 (unknown-machine-state))
         (state-2 state-2)
         (netlist (chip$netlist))))))


(prove-lemma fm9001-machine-statep-p-map-up-initialized-machine-state (rewrite)
  (fm9001-machine-statep (p-map-up (initialized-machine-state)))
  ((disable fm9001-machine-statep map-up initialized-machine-state)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    Refinements
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn v-fourp (x)
  (if (nlistp x)
      (equal x nil)
    (and (fourp (car x))
         (v-fourp (cdr x)))))
```

```
(defn all-xs (vec)
  (if (nlistp vec)
      (equal vec nil)
    (and (equal (car vec) (X))
         (all-xs (cdr vec)))))

;; The following two were enabled in the context where many
;; of the following events were originally developed.


(enable b-knownp)

(enable b-approx)


(prove-lemma all-xs-approximates (rewrite)
  (implies (and (all-xs vec1)
                (equal (length vec1) (length vec2))
                (properp vec2))
           (v-approx vec1 vec2))
  ((enable v-approx)))


(prove-lemma all-xs-make-list (rewrite)
  (all-xs (make-list n (x)))
  ((enable make-list)))


(defn memory-v-fourp (n width mem)
  (cond ((stubp mem)
         (and (v-fourp (stub-guts mem))
              (equal (length (stub-guts mem))
                     width)))
        ((zerop n)
         (cond ((ramp mem)
                (and (v-fourp (ram-guts mem))
                     (equal (length (ram-guts mem))
                            width)))
               ((romp mem)
                (and (v-fourp (rom-guts mem))
                     (equal (length (rom-guts mem))
                            width)))
               (t f)))
        (t (and (listp mem)
                (memory-v-fourp (sub1 n)
                                width
                                (car mem))
                (memory-v-fourp (sub1 n)
                                width
                                (cdr mem))))))
```

```
(defn new-machine-state-invariant (machine-state)
  (let
    ((regs              (car machine-state))
     (flags             (cadr machine-state))
     (a-reg             (caddr machine-state))
     (b-reg             (cadddr machine-state))
     (i-reg             (caddddr machine-state))
     (data-out          (cadddddr machine-state))
     (addr-out          (caddddddr machine-state))
     (last-reset-       (cadddddddr machine-state))
     (last-dtack-       (caddddddddr machine-state))
     (last-hold-        (cadddddddddr machine-state))
     (pc-reg            (caddddddddddr machine-state))
     (cntl-state        (cadddddddddddr machine-state)))
    (let
      ((regs-regs    (car regs))
       (regs-we      (cadr regs))
       (regs-data    (caddr regs))
       (regs-address (cadddr regs)))
      (and
       (equal (length machine-state) 12)
       (properp machine-state)
       (equal (length regs) 4)
       (properp regs)
       (all-ramp-mem 4 regs-regs)
       (fourp regs-we)
       (memory-v-fourp 4 32 regs-regs)
       (v-fourp regs-data) (equal (length regs-data) 32)
       (v-fourp regs-address) (equal (length regs-address) 4)
       (v-fourp flags) (equal (length flags) 4)
       (v-fourp a-reg) (equal (length a-reg) 32)
       (v-fourp b-reg) (equal (length b-reg) 32)
       (v-fourp i-reg) (equal (length i-reg) 32)
       (v-fourp data-out) (equal (length data-out) 32)
       (v-fourp addr-out) (equal (length addr-out) 32)
       (fourp last-reset-)
       (fourp last-dtack-)
       (fourp last-hold-)
       (v-fourp pc-reg) (equal (length pc-reg) 4)
       (v-fourp cntl-state) (equal (length cntl-state) 40)))))


(prove-lemma s-approx-make-list (rewrite)
  (implies (and (equal (length vec1) (length vec2))
                (v-fourp vec2)
                (all-xs vec1))
           (s-approx vec1 vec2))
  ((enable s-approx)))


(prove-lemma v-fourp-implies-properp (rewrite)
  (implies (v-fourp x)
           (properp x)))
```

```
(prove-lemma listp-implies-not-fourp (rewrite)
  (implies (listp x)
           (not (fourp x))))


(prove-lemma ramp-implies-not-fourp (rewrite)
  (implies (ramp x)
           (not (fourp x))))


(prove-lemma romp-implies-not-fourp (rewrite)
  (implies (romp x)
           (not (fourp x))))


(prove-lemma stubp-implies-not-fourp (rewrite)
  (implies (stubp x)
           (not (fourp x))))


(prove-lemma b-approx-x (rewrite)
  (b-approx (x) y))


(prove-lemma machine-state-invariant-implies-s-approx-lemma ()
  (implies (and (new-machine-state-invariant state)
                (equal state
                       (list (list s1-1 s1-2 s1-3 s1-4)
                             s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12)))
           (s-approx (unknown-machine-state) state))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable new-machine-state-invariant
           b-approx-x
           all-ramp-mem
           listp-implies-not-fourp
           ramp-implies-not-fourp
           romp-implies-not-fourp
           stubp-implies-not-fourp
           *1*fourp
           v-fourp-implies-properp
           all-xs-approximates
           length *1*length
           length-cons
           all-xs
           all-xs-make-list
           s-approx-make-list
           s-approx
           *1*unknown-machine-state
           ram-guts-ram
           rom-guts-rom
           memory-v-fourp
           stub-guts-stub)))
```

```
(prove-lemma machine-state-invariant-implies-s-approx-lemma-2 ()
  (let ((s1-1 (caar state))
        (s1-2 (cadar state))
        (s1-3 (caddar state))
        (s1-4 (cadddar state))
        (s2  (cadr state))
        (s3  (caddr state))
        (s4  (cadddr state))
        (s5  (caddddr state))
        (s6  (cadddddr state))
        (s7  (caddddddr state))
        (s8  (cadddddddr state))
        (s9  (caddddddddr state))
        (s10 (cadddddddddr state))
        (s11 (caddddddddddr state))
        (s12 (cadddddddddddr state)))
    (implies (and (equal (length state) 12)
                  (properp state)
                  (equal (length (car state)) 4)
                  (properp (car state)))
             (equal state
                    (list (list s1-1 s1-2 s1-3 s1-4)
                          s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable length length-bottom properp)))


(prove-lemma machine-state-invariant-implies-s-approx-lemma-3 ()
  (implies (new-machine-state-invariant state)
           (and (equal (length state) 12)
                (properp state)
                (equal (length (car state)) 4)
                (properp (car state))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable new-machine-state-invariant)))
```

```
(prove-lemma machine-state-invariant-implies-s-approx ()
  (implies (new-machine-state-invariant state)
           (s-approx (unknown-machine-state) state))
  ((disable-theory t)
   (enable-theory ground-zero)
   (use (machine-state-invariant-implies-s-approx-lemma-2)
        (machine-state-invariant-implies-s-approx-lemma-3)
        (machine-state-invariant-implies-s-approx-lemma
         (s1-1 (caar state))
         (s1-2 (cadar state))
         (s1-3 (caddar state))
         (s1-4 (cadddar state))
         (s2  (cadr state))
         (s3  (caddr state))
         (s4  (cadddr state))
         (s5  (caddddr state))
         (s6  (cadddddr state))
         (s7  (caddddddr state))
         (s8  (cadddddddr state))
         (s9  (caddddddddr state))
         (s10 (cadddddddddr state))
         (s11 (caddddddddddr state))
         (s12 (cadddddddddddr state))))))


(prove-lemma good-s-opener (rewrite)
  (and (implies (listp s)
                (equal (good-s s)
                       (and (good-s (car s))
                            (good-s (cdr s)))))
       (implies (ramp s)
                (equal (good-s s)
                       (and (equal (length (ram-guts s))
                                   (mem-width))
                            (properp (ram-guts s)))))
       (implies (romp s)
                (equal (good-s s)
                       (and (equal (length (rom-guts s))
                                   (mem-width))
                            (properp (rom-guts s)))))
       (implies (stubp s)
                (equal (good-s s)
                       (and (equal (length (stub-guts s))
                                   (mem-width))
                            (properp (stub-guts s)))))
       (implies (fourp s)
                (good-s s)))
  ((enable good-s)))


(prove-lemma memory-v-fourp-implies-good-s (rewrite)
  (implies (memory-v-fourp n 32 v)
           (good-s v)))
```

```
(prove-lemma v-fourp-is-good-s (rewrite)
  (implies (v-fourp v)
           (good-s v))
  ((enable good-s)))


(prove-lemma new-machine-state-invariant-implies-good-s ()
  (implies (new-machine-state-invariant state)
           (good-s state))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable good-s-opener new-machine-state-invariant
           v-fourp-is-good-s length-cons length-1 length-bottom
           properp-cons properp-nlistp
           *1*good-s
           memory-v-fourp-implies-good-s)))


(prove-lemma xs-suffice-for-reset-chip-final-state-for-any-unknown-state ()
  (let ((n (sub1 (length (reset-sequence-chip-1)))))

    (let ((final-1 (cadr (nth n (simulate 'chip
                                          (reset-sequence-chip-1)
                                          (unknown-machine-state)
                                          (chip$netlist)))))
          (final-2 (cadr (nth n (simulate 'chip
                                          (reset-sequence-chip-1)
                                          state
                                          (chip$netlist))))))

      (implies (new-machine-state-invariant state)
               (equal final-1 final-2))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable final-state)
   (use (xs-suffice-for-reset-chip-lemma-instance
         (final-1 (car (final-state)))
         (state-2 state))
        (simulate-reset-chip-final-state)
        (machine-state-invariant-implies-s-approx)
        (new-machine-state-invariant-implies-good-s))))

;; The following is a main theorem.  It follows readily from
;; XS-SUFFICE-FOR-RESET-CHIP-LEMMA-INSTANCE together with the theorem
;; SIMULATE-RESET-CHIP-FINAL-STATE, which essentially tells us that
;; when we simulate from the (UNKNOWN-MACHINE-STATE) we get to
;; (INITIALIZED-MACHINE-STATE).  The main improvement of this theorem
;; over XS-SUFFICE-FOR-RESET-CHIP-LEMMA-INSTANCE is that the
;; hypothesis here avoids the notion of ``approximation'' in favor
;; of a simple invariant on the structure of the state (see
;; NEW-MACHINE-STATE-INVARIANT-IS-NON-TRIVIAL and
;; NEW-MACHINE-STATE-INVARIANT-IMPLIES-MACHINE-STATE-INVARIANT
;; below).
```

```
(prove-lemma
  xs-suffice-for-reset-chip-final-state-for-any-unknown-state-better
  (rewrite)
  (let ((n (sub1 (length (reset-sequence-chip-1)))))
    (implies (new-machine-state-invariant state)
             (equal (cadr (nth n (simulate 'chip
                                            (reset-sequence-chip-1)
                                            state
                                            (chip$netlist))))
                    (initialized-machine-state))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable final-state)
   (use (xs-suffice-for-reset-chip-final-state-for-any-unknown-state)
        (simulate-reset-chip-final-state))))


(prove-lemma new-machine-state-invariant-is-non-trivial ()
  (new-machine-state-invariant (unknown-machine-state))
  ((disable-theory t)
   (enable *1*new-machine-state-invariant
           *1*unknown-machine-state)))


(prove-lemma memory-v-fourp-implies-memory-properp (rewrite)
  (implies (memory-v-fourp n width mem)
           (memory-properp n width mem))
  ((enable memory-properp)))


(prove-lemma
  new-machine-state-invariant-implies-machine-state-invariant
  (rewrite)
  (implies (new-machine-state-invariant machine-state)
           (machine-state-invariant machine-state))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable new-machine-state-invariant machine-state-invariant
           v-fourp-implies-properp
           memory-v-fourp-implies-memory-properp)))

;==============================

;; Consider the lemma FM9001=CHIP-SYSTEM from "proofs.events".  We can
;; replace the FM9001-STATEP hypothesis in that lemma with a
;; MEMORY-OKP hypothesis for the class of machine states we are
;; considering, i.e., those obtained at the end of the reset sequence.
```

```
(prove-lemma fm9001-statep-implies-memory-ok-p-instance ()
  (let ((state (map-up (list (initialized-machine-state) machine-memory))))
    (implies (memory-okp 32 32 (cadr state))
             (fm9001-statep state)))
  ((enable *1*initialized-machine-state *1*fm9001-machine-statep)))

;; Just a lemma along the way to FM9001=CHIP-SYSTEM-TRUE-AFTER-RESET.


(prove-lemma fm9001=chip-system-true-after-reset-lemma (rewrite)
  (let ((state (map-up (list (initialized-machine-state) machine-memory))))
    (implies (and (chip-system& netlist)
                  ;;(fm9001-statep state)
                  (memory-okp 32 32 (cadr state))
                  (chip-system-operating-inputs-p
                   inputs
                   (total-microcycles (map-down state)
                                      (map-up-inputs inputs)
                                      n))
                  (operating-inputs-p
                   (map-up-inputs inputs)
                   (total-microcycles (map-down state)
                                      (map-up-inputs inputs)
                                      n)))
             (equal (fm9001 state n)
                    (map-up
                     (simulate-dual-eval-2
                      'chip-system inputs
                      (map-down state)
                      netlist
                      (total-microcycles (map-down state)
                                         (map-up-inputs inputs)
                                         n))))))
  ((use (fm9001=chip-system
         (state (map-up (list (initialized-machine-state) machine-memory))))
        (fm9001-statep-implies-memory-ok-p-instance))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable fm9001-statep
           properp length
           length-cons length-bottom
           properp-cons properp-nlistp)))


(prove-lemma map-down-inverts-map-up (rewrite)
  (let ((machine-memory
         (list memory 0 0 0 t t (make-list 32 f) (make-list 32 (x)))))
    (let ((state (map-up (list (initialized-machine-state) machine-memory))))
      (implies (memory-okp 32 32 (cadr state))
               (equal (map-down state)
                      (list (initialized-machine-state) machine-memory))))))
```

```
(prove-lemma cadr-map-up ()
  (equal (cadr (map-up (list s mem)))
         (car mem)))

;; Here is a statement of FM9001=CHIP-SYSTEM, which, when combined
;; with the reset lemma
;; XS-SUFFICE-FOR-RESET-CHIP-FINAL-STATE-FOR-ANY-UNKNOWN-STATE-BETTER
;; above, says that the chip implements its specification assuming
;; that we attach the memory after the reset process.  This is a
;; general statement; without a particular memory design, it seems
;; difficult to come up with a statement that is much ``better'' than
;; this one.


(prove-lemma fm9001=chip-system-true-after-reset (rewrite)
  (let ((machine-memory
         (list memory 0 0 0 t t (make-list 32 f) (make-list 32 (x)))))
    (let ((state (map-up (list (initialized-machine-state) machine-memory)))
          (low-state (list (initialized-machine-state) machine-memory)))
      (implies (and (chip-system& netlist)
                    (memory-okp 32 32 memory)
                    (chip-system-operating-inputs-p
                     inputs
                     (total-microcycles low-state
                                        (map-up-inputs inputs)
                                        n))
                    (operating-inputs-p
                     (map-up-inputs inputs)
                     (total-microcycles low-state
                                        (map-up-inputs inputs)
                                        n)))
               (equal (fm9001 state n)
                      (map-up
                       (simulate-dual-eval-2
                        'chip-system inputs
                        low-state
                        netlist
                        (total-microcycles low-state
                                           (map-up-inputs inputs)
                                           n)))))))
  ((use (fm9001=chip-system-true-after-reset-lemma
         (machine-memory
          (list memory 0 0 0 t t (make-list 32 f) (make-list 32 (x)))))
        (cadr-map-up
         (s (initialized-machine-state))
         (mem (list memory 0 0 0 t t (make-list 32 f) (make-list 32 (x))))))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable map-down-inverts-map-up cadr-map-up)))
```

```
(disable map-down-inverts-map-up)

;; Let us compensate now for the corresponding two enables
;; appearing above.


(disable b-knownp)

(disable b-approx)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    Final Statements
;;;
;;;    Below are our most general statements about the correctness of the
;;;    FM9001 microprocessor.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;  Some of the earlier lemmas were written in terms of SIMULATE.  Since we
;;;  are only interested in the final state, we show that SIMULATE contains
;;;  SIMULATE-DUAL-EVAL-2.


(prove-lemma simulate-contains-simulate-dual-eval-2 ()
  (implies
   (and (not (zerop n))
        (leq n (length inputs)))
   (equal (cadr (nth (sub1 n) (simulate fn inputs state netlist)))
          (simulate-dual-eval-2 fn inputs state netlist n)))
  ;;Hint
  ((induct (simulate-dual-eval-2 fn inputs state netlist n))
   (enable simulate simulate-dual-eval-2)))


(prove-lemma length-reset-sequence-chip-1 (rewrite)
  (equal (length (reset-sequence-chip-1))
         20))

;;;   RESET-CHIP shows that CHIP can be reset from a completely unknown state.
```

```
(prove-lemma reset-chip ()
  (equal (simulate-dual-eval-2
          'chip
          (reset-sequence-chip-1)
          (unknown-machine-state)
          (chip$netlist)
          (length (reset-sequence-chip-1)))
         (initialized-machine-state))
  ;;Hint
  ((use (simulate-contains-simulate-dual-eval-2
         (n (length (reset-sequence-chip-1)))
         (inputs (reset-sequence-chip-1))
         (fn 'chip)
         (state (unknown-machine-state))
         (netlist (chip$netlist)))
        (simulate-reset-chip-final-state))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable final-state length-reset-sequence-chip-1)))

;;;   RESET-CHIP-FROM-ANY-STATE shows that the machine can be reset from any
;;;   state of the proper shape.
```

```
(prove-lemma reset-chip-from-any-state ()
  (implies
   (and (s-approx (unknown-machine-state) any-state)
        (good-s any-state))
   (equal (simulate-dual-eval-2
           'chip
           (reset-sequence-chip-1)
           any-state
           (chip$netlist)
           (length (reset-sequence-chip-1)))
          (initialized-machine-state)))
  ;;Hint
  ((use (simulate-contains-simulate-dual-eval-2
         (n (length (reset-sequence-chip-1)))
         (inputs (reset-sequence-chip-1))
         (fn 'chip)
         (state (unknown-machine-state))
         (netlist (chip$netlist)))
        (simulate-contains-simulate-dual-eval-2
         (n (length (reset-sequence-chip-1)))
         (inputs (reset-sequence-chip-1))
         (fn 'chip)
         (state any-state)
         (netlist (chip$netlist)))
        (simulate-reset-chip-final-state)
        (xs-suffice-for-reset-chip-lemma-instance
         (state-2 any-state)))
   (disable-theory t)
   (enable-theory ground-zero)
   (enable final-state length-reset-sequence-chip-1)))

;;;  CHIP-SYSTEM=FM9001-INTERPRETER$AFTER-RESET is the same as the lemma
;;;  chip-system=fm9001-interpreter in "proofs.events", except that it's
;;;  specialized to the case where the initial state is made up of the chip
;;;  state after reset (i.e., (INITIALIZED-MACHINE-STATE)) together with an
;;;  appropriate machine memory.  Thus, the hypothesis
;;;  (fm9001-state-structure state) has been omitted and the rather
;;;  elaborate hypothesis (macrocycle-invariant state) has been replaced by
;;;  the much weaker hypothesis (memory-okp 32 32 memory).  We can do this
;;;  because (INITIALIZED-MACHINE-STATE) satisfies the processor state
;;;  portions of these two hypotheses.
```

```
(prove-lemma chip-system=fm9001-interpreter$after-reset (rewrite)
  (let
    ((state
      (list
       (initialized-machine-state)
       (list memory 0 clock 0 (x) t (make-list 32 f) (make-list 32 (x))))))
    (let
      ((rtl-inputs (map-up-inputs inputs)))
      (let
        ((clock-cycles (total-microcycles state rtl-inputs instructions)))
        (implies
         (and (chip-system& netlist)
              (memory-okp 32 32 memory)
              (chip-system-operating-inputs-p inputs clock-cycles)
              (operating-inputs-p rtl-inputs clock-cycles))
         (equal (map-up (simulate-dual-eval-2
                         'chip-system inputs state netlist clock-cycles))
                (fm9001-interpreter
                 (map-up state) (make-list 4 t) instructions))))))
  ;;Hint
  ((use (chip-system=fm9001-interpreter
         (state
          (list
           (initialized-machine-state)
           (list memory 0 clock 0 (x) t (make-list 32 f) (make-list 32 (x)))))
         (pc (make-list 4 t))))
   (disable chip-system=fm9001-interpreter)))
```

14.69 "well-formed-fm9001.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    WELL-FORMED-FM9001.EVENTS
;;;
;;;  These events guarantee that the FM9001 netlists are well formed.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;;  We first have to compile the definitions or this proof takes
;;;  too long.


(compile-uncompiled-defns* "a-weird-file-name")


(prove-lemma chip-well-formed ()
  (and (equal (top-level-predicate (chip-module$netlist))
              T)
       (equal (top-level-predicate (chip$netlist))
              T))
  ((disable top-level-predicate chip-module$netlist chip$netlist)))


(prove-lemma chip-well-formed-after-indexed-names-removed ()
  (and (equal (top-level-predicate (lisp-netlist (chip-module$netlist)))
              T)
       (equal (top-level-predicate (lisp-netlist (chip$netlist)))
              T))
  ((disable top-level-predicate lisp-netlist
            chip-module$netlist chip$netlist)))


;;  The "simple" version of the predicate does not permit tri-state
;;  circuits, thus we can only check the "body" of the chip.


(prove-lemma chip-well-formed-simple ()
  (and (equal (top-level-predicate-simple (chip-module$netlist))
              T)
       (equal (top-level-predicate-simple
               (lisp-netlist (chip-module$netlist)))
              T))
  ((disable top-level-predicate chip-module$netlist)))
```

14.70 "math-enable.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    MATH-ENABLE.EVENTS
;;;
;;;
;;;  This file is to enable the sets, naturals, and integers
;;;  libraries that were previously disabled.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(better-disable-back-to b-knownp)


(enable-theory math-theory)
```

14.71 "alu-interpretation.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    ALU-INTERPRETATION.EVENTS
;;;
;;;  Proofs of correctness of V-ALU.  The lemmas proven here mirror the lemmas
;;;  proven in the FM8501/FM8502 efforts.  These lemmas are actually proofs
;;;  about the behavioral level specification; that is, these lemmas show that
;;;  the ALU correctly performs various arithmetic operations when the bit
;;;  vector inputs are viewed as integers and natural numbers.
;;;
;;;  Proofs by Matt Kaufmann
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;; Table of contents

;; I.   *PRELIMINARIES*
;;      A.  V-BUF

;; II.  *NATURALS*
;;      A. USEFUL LEMMAS for naturals case
;;      B. V-ADDER
;;      C. V-INC
;;      D. V-SUBTRACTER
;;      E. V-DEC
;;      F. V-LSR
;;      G.  CORRECTNESS -- natural number case

;; III. *INTEGERS*
;;      A. USEFUL LEMMAS for integers case
;;      B. V-ADDER
;;      C. V-INC
;;      D. V-SUBTRACTER
;;      E. V-DEC
;;      F. V-ASR
;;      G. V-NEG
;;      H.  CORRECTNESS -- integers case

;; IV.  *FINAL DISABLES*

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;               *PRELIMINARIES*               ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defn b-to-nat (c)
  (if c 1 0))


(disable v-adder-output=v-sum)

(disable v-adder-carry-out=v-carry)


(prove-lemma bvp-cdr (rewrite)
  ;; for speed
  (implies (and (bvp x) (listp x))
           (bvp (cdr x))))


(deftheory integer-metas
  (correctness-of-cancel-ineg
   correctness-of-cancel-iplus
   correctness-of-cancel-iplus-ilessp
   correctness-of-cancel-itimes
   correctness-of-cancel-itimes-ilessp
   correctness-of-cancel-itimes-factors
   correctness-of-cancel-itimes-ilessp-factors
   correctness-of-cancel-factors-0
   correctness-of-cancel-factors-ilessp-0
   correctness-of-cancel-ineg-terms-from-equality
   correctness-of-cancel-ineg-terms-from-inequality
   ))


(disable correctness-of-cancel-ineg)

(disable correctness-of-cancel-iplus)

(disable correctness-of-cancel-iplus-ilessp)

(disable correctness-of-cancel-itimes)

(disable correctness-of-cancel-itimes-ilessp)

(disable correctness-of-cancel-itimes-factors)

(disable correctness-of-cancel-itimes-ilessp-factors)

(disable correctness-of-cancel-factors-0)

(disable correctness-of-cancel-factors-ilessp-0)

(disable correctness-of-cancel-ineg-terms-from-equality)

(disable correctness-of-cancel-ineg-terms-from-inequality)
```

```
(deftheory naturals-metas
  (correctness-of-cancel-equal-plus
   correctness-of-cancel-difference-plus
   correctness-of-cancel-lessp-plus
   correctness-of-cancel-lessp-times
   correctness-of-cancel-equal-times
   correctness-of-cancel-quotient-times))


(disable correctness-of-cancel-equal-plus)

(disable correctness-of-cancel-difference-plus)

(disable correctness-of-cancel-lessp-plus)

(disable correctness-of-cancel-lessp-times)

(disable correctness-of-cancel-equal-times)

(disable correctness-of-cancel-quotient-times)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                   V-BUF                     ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn v-alu-nat-int-buf (a)
  (cvzbv f f a))


(prove-lemma v-alu-correct-nat-int-buf (rewrite)
  (implies (bvp a)
           (equal (cvzbv f f (v-buf a))
                  (v-alu-nat-int-buf a))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                 *NATURALS*                  ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;       USEFUL LEMMAS for naturals case       ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;
;;; lemmas for v-alu-correct-nat-adder-carry-out
;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(prove-lemma nth-length-v (rewrite)
  (implies (and (bvp bv)
                (equal (length bv) (add1 n)))
           (equal (nth n bv)
                  (not (lessp (v-to-nat bv)
                              (exp 2 n)))))
  ((enable v-to-nat nth)
   (induct (nth n bv))))

;;;;;;;;;;;;;;;;;;;;;;;;;
;;; lemmas for v-alu-correct-nat-adder-output
;;;;;;;;;;;;;;;;;;;;;;;;;

;;; In order to prove v-to-nat-firstn, we need some
;;; lemmas about remainder, which I'll do myself rather than adapting
;;; them from bv-to-nat-of-trunc in fm8502, because I don't want to
;;; work on re-creating that environment.


(prove-lemma remainder-plus-multiple nil
  (implies (and (lessp z1 y1)
                (equal (remainder a y1) 0))
           (equal (remainder (plus z1 a)
                             y1)
                  (fix z1))))


(prove-lemma divides-plus-plus (rewrite)
  (implies (equal (remainder x y) 0)
           (equal (remainder (plus x x) (plus y y)) 0)))


(prove-lemma v-to-nat-firstn-hack1 (rewrite)
  (implies (lessp z y)
           (equal (remainder (plus z z (times w y) (times w y))
                             (plus y y))
                  (plus z z)))
  ((use (remainder-plus-multiple
         (y1 (plus y y))
         (z1 (plus z z))
         (a (plus (times w y) (times w y)))))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable divides-plus-plus)))
```

```
(prove-lemma remainder-add1-plus-multiple nil
  (implies (and (lessp (add1 z1) y1)
                (equal (remainder a y1) 0))
           (equal (remainder (add1 (plus z1 a))
                             y1)
                  (add1 z1)))
  ((use (remainder-plus-multiple
         (z1 (add1 z1))))))


(prove-lemma v-to-nat-firstn-hack2 (rewrite)
  (implies (lessp z y)
           (equal (remainder (add1 (plus z z (times d y) (times d y)))
                             (plus y y))
                  (add1 (plus z z))))
  ((use (remainder-add1-plus-multiple
         (y1 (plus y y))
         (z1 (plus z z))
         (a (plus (times d y) (times d y)))))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable divides-plus-plus)))


(prove-lemma v-to-nat-firstn (rewrite)
  ;; adapted from fm8502 lemma bv-to-nat-of-trunc
  (equal (v-to-nat (firstn n v))
         (remainder (v-to-nat v) (exp 2 n)))
  ((enable v-to-nat firstn)))


(disable v-to-nat-firstn-hack1)

(disable v-to-nat-firstn-hack2)


(prove-lemma v-to-nat-of-nat-to-v-hack (rewrite)
  (implies
   (lessp v (exp 2 x))
   (equal (remainder (add1 (plus v v
                                 (times w (exp 2 x))
                                 (times w (exp 2 x))))
                     (plus (exp 2 x) (exp 2 x)))
          (add1 (plus v v))))
  ((use (remainder-plus-multiple
         (y1 (plus (exp 2 x) (exp 2 x)))
         (z1 (add1 (plus v v)))
         (a (plus (times w (exp 2 x)) (times w (exp 2 x))))))
   (disable-theory t)
   (enable-theory ground-zero naturals)
   (enable divides-plus-plus)))
```

```
(prove-lemma v-to-nat-of-nat-to-v (rewrite)
  (equal (v-to-nat (nat-to-v n len))
         (remainder n (exp 2 len)))
  ((enable v-to-nat nat-to-v)
   (induct (nat-to-v n len))))


(prove-lemma remainder-plus-x-x-2 (rewrite)
             (equal (remainder (plus x x) 2)
                    0)
             ((induct (plus x q))))


(prove-lemma quotient-plus-x-x-2 (rewrite)
             (equal (quotient (plus x x) 2)
                    (fix x))
             ((induct (plus x q))))


(prove-lemma nat-to-v-plus-x-x (rewrite)
  (equal (nat-to-v (plus x x) (add1 n))
         (cons f (nat-to-v x n)))
  ((enable nat-to-v)))


(prove-lemma firstn-add1-cons (rewrite)
  (equal (firstn (add1 n) (cons a x))
         (cons a (firstn n x)))
  ((enable firstn)))


(prove-lemma firstn-zerop (rewrite)
  (implies (zerop n)
           (equal (firstn n x)
                  nil))
  ((enable firstn)))


(prove-lemma firstn-nlistp (rewrite)
  (implies (nlistp x)
           (equal (firstn n x)
                  nil))
  ((enable firstn)))


(prove-lemma nat-to-v-of-v-to-nat-general (rewrite)
  (implies (and (bvp v)
                (not (lessp (length v) n)))
           (equal (nat-to-v (v-to-nat v) n)
                  (firstn n v)))
  ((enable v-to-nat nat-to-v)
   (induct (firstn n v))))
```

```
(prove-lemma firstn-length (rewrite)
  (implies (properp x)
           (equal (firstn (length x) x)
                  x))
  ((enable firstn)))


(prove-lemma nat-to-v-of-v-to-nat (rewrite)
  (implies (bvp v)
           (equal (nat-to-v (v-to-nat v) (length v))
                  v)))


(prove-lemma nat-to-v-zerop (rewrite)
  (implies (zerop len)
           (equal (nat-to-v n len) nil))
  ((enable nat-to-v)))


(prove-lemma equal-nat-to-v-inverter-hack1 (rewrite)
  (implies (and (numberp len)
                (equal v (nat-to-v n len)))
           (equal (equal (v-to-nat v)
                         (remainder n (exp 2 len)))
                  t)))


(prove-lemma equal-nat-to-v-inverter-hack2-lemma nil
  (implies (and (bvp v)
                (equal (v-to-nat v)
                       (remainder n (exp 2 len))))
           (equal (equal (nat-to-v (v-to-nat v) len)
                         (nat-to-v (remainder n (exp 2 len))
                                   len))
                  t)))


(prove-lemma nat-to-v-remainder (rewrite)
  (equal (nat-to-v (remainder n (exp 2 len))
                   len)
         (nat-to-v n len))
  ((enable nat-to-v)))


(prove-lemma equal-nat-to-v-inverter-hack2 (rewrite)
  (implies (and (bvp v)
                (equal (v-to-nat v)
                       (remainder n (exp 2 (length v)))))
           (equal (equal v (nat-to-v n (length v)))
                  t))
  ((use (equal-nat-to-v-inverter-hack2-lemma
         (len (length v))))))
```

```
(prove-lemma equal-nat-to-v-inverter (rewrite)
  (iff (equal v (nat-to-v n len))
       (and (bvp v)
            (equal (length v) (fix len))
            (equal (v-to-nat v) (remainder n (exp 2 len))))))


(disable equal-nat-to-v-inverter-hack1)

(disable equal-nat-to-v-inverter-hack2)

;;;;;;;;;;;;;;;;;;;;;;;;;
;;; lemmas for v-alu-correct-nat-subtracter-carry-out
;;;;;;;;;;;;;;;;;;;;;;;;;


(prove-lemma v-to-nat-v-not-lemma nil
  (implies (bvp a)
           (equal (plus (v-to-nat a) (v-to-nat (v-not a)))
                  (sub1 (exp 2 (length a)))))
  ((enable v-to-nat v-not)))


(prove-lemma v-to-nat-v-not (rewrite)
  (implies (bvp a)
           (equal (v-to-nat (v-not a))
                  (sub1 (difference (exp 2 (length a))
                                    (v-to-nat a)))))
  ((use (v-to-nat-v-not-lemma))))


(prove-lemma lessp-v-to-nat-exp (rewrite)
  (implies (bvp a)
           (lessp (v-to-nat a)
                  (exp 2 (length a))))
  ((enable v-to-nat length)))


(prove-lemma equal-iff (rewrite)
  (implies (and (boolp x) (boolp y))
           (equal (equal x y)
                  (iff x y)))
  ((enable boolp)))


(disable equal-iff)

;;;;;;;;;;;;;;;;;;;;;;;;;
;;; lemmas for v-alu-correct-nat-subtracter-output
;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(prove-lemma v-alu-correct-nat-subtracter-output-lemma (rewrite)
  (implies (and (lessp a-nat bound)
                (lessp b-nat bound))
           (equal (plus b-nat
                        (sub1 (difference bound a-nat)))
                  (sub1 (difference (plus b-nat bound)
                                    a-nat)))))


(prove-lemma lessp-v-to-nat-exp-rewrite (rewrite)
  ;; should be useful when len is (length b), arising from hyp. (bv2p a b)
  (implies (and (bvp a) (equal (length a) len))
           (equal (lessp (v-to-nat a)
                         (exp 2 len))
                  t)))

;;;;;;;;;;;;;;;;;;;;;;;;
;;; lemmas for v-alu-correct-nat-lsr-output
;;;;;;;;;;;;;;;;;;;;;;;;


(prove-lemma not-lessp-quotient (rewrite)
  (not (lessp n (quotient n k))))


(prove-lemma v-to-nat-append (rewrite)
  (implies (and (bvp a) (bvp b))
           (equal (v-to-nat (append a b))
                  (plus (v-to-nat a)
                        (times (exp 2 (length a))
                               (v-to-nat b)))))
  ((enable v-to-nat)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                  V-ADDER                   ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn v-alu-nat-adder-output (c a b)
  ;; specification function for v-adder-output
  (nat-to-v (remainder (plus (b-to-nat c) (v-to-nat a) (v-to-nat b))
                       (exp 2 (length a)))
            (length a)))


(defn v-alu-nat-adder-carry-out (c a b)
  ;; specification function for v-adder-carry-out
  (not (lessp (plus (b-to-nat c) (v-to-nat a) (v-to-nat b))
              (exp 2 (length a)))))
```

```
(defn v-alu-nat-adder (c a b)
  ;; specification function for cvzbv-v-adder
  (cvzbv (v-alu-nat-adder-carry-out c a b)
         ;; we don't have anything special to say about overflows in
         ;; the natural number case
         (v-adder-overflowp         c a b)
         (v-alu-nat-adder-output    c a b)))


(prove-lemma v-alu-correct-nat-adder-carry-out (rewrite)
  (implies (bv2p a b)
           (equal (v-adder-carry-out c a b)
                  (v-alu-nat-adder-carry-out c a b))))


(prove-lemma v-alu-correct-nat-adder-output (rewrite)
  (implies (bv2p a b)
           (equal (v-adder-output c a b)
                  (v-alu-nat-adder-output c a b))))


(prove-lemma v-alu-correct-nat-adder (rewrite)
  (implies (bv2p a b)
           (equal (cvzbv-v-adder c a b)
                  (v-alu-nat-adder c a b)))
  ((disable v-adder-carry-out v-adder-overflowp
            v-adder-output v-alu-nat-adder-carry-out
            v-alu-nat-adder-output)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                    V-INC                    ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn v-alu-nat-inc (a)
  ;; specification function for cvzbv-inc -- it seems quite reasonable
  ;; to use the specification function for the adder to express this
  ;; property
  (v-alu-nat-adder t a (nat-to-v 0 (length a))))


(prove-lemma v-alu-correct-nat-inc (rewrite)
  (implies (bvp a)
           (equal (cvzbv-inc a)
                  (v-alu-nat-inc a)))
  ((disable v-alu-nat-adder cvzbv-v-adder)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                 V-SUBTRACTER                ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defn v-alu-nat-subtracter-output (c a b)
  ;; specification function for v-subtracter-output
  ;; The idea here is that it's b - (c + a), but since we want
  ;; to stay in the realm of the natural numbers, we add
  ;; 2**n to be in order to be sure that this is a "proper"
  ;; subtraction.
  (nat-to-v (remainder (difference (plus (v-to-nat b) (exp 2 (length a)))
                                   (plus (v-to-nat a) (b-to-nat c)))
                       (exp 2 (length a)))
            (length a)))


(defn v-alu-nat-subtracter-carry-out (c a b)
  ;; specification function for v-subtracter-carry-out
  (lessp (v-to-nat b)
         (plus (v-to-nat a) (b-to-nat c))))


(defn v-alu-nat-subtracter (c a b)
  ;; specification function for cvzbv-v-subtracter
  (cvzbv (v-alu-nat-subtracter-carry-out c a b)
         ;; we don't have anything special to say about overflows in
         ;; the natural number case
         (v-subtracter-overflowp         c a b)
         (v-alu-nat-subtracter-output    c a b)))


(prove-lemma v-alu-correct-nat-subtracter-carry-out (rewrite)
  (implies (bv2p a b)
           (equal (v-subtracter-carry-out c a b)
                  (v-alu-nat-subtracter-carry-out c a b)))
  ((enable equal-iff)))


(prove-lemma v-alu-correct-nat-subtracter-output (rewrite)
  (implies (bv2p a b)
           (equal (v-subtracter-output c a b)
                  (v-alu-nat-subtracter-output c a b))))


(prove-lemma v-alu-correct-nat-subtracter (rewrite)
  (implies (bv2p a b)
           (equal (cvzbv-v-subtracter c a b)
                  (v-alu-nat-subtracter c a b)))
  ((disable v-subtracter-carry-out v-subtracter-overflowp
            v-subtracter-output v-alu-nat-subtracter-carry-out
            v-alu-nat-subtracter-output)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                    V-DEC                    ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defn v-alu-nat-dec (a)
  ;; specification function for cvzbv-dec -- it seems quite reasonable
  ;; to use the specification function for the adder to express this
  ;; property
  (v-alu-nat-subtracter t (nat-to-v 0 (length a)) a))


(prove-lemma v-alu-correct-nat-dec (rewrite)
  (implies (bvp a)
           (equal (cvzbv-dec a)
                  (v-alu-nat-dec a)))
  ((disable v-alu-nat-subtracter cvzbv-v-subtracter)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                      V-LSR                     ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn v-alu-nat-lsr-output (a)
  (nat-to-v (quotient (v-to-nat a) 2)
            (length a)))


(defn v-alu-nat-lsr-carry-out (a)
  (not (equal (remainder (v-to-nat a) 2) 0)))


(defn v-alu-nat-lsr (a)
  (cvzbv (v-alu-nat-lsr-carry-out a)
         ;; we don't have anything special to say about overflows in
         ;; the natural number case
         f
         (v-alu-nat-lsr-output a)))


(prove-lemma v-alu-correct-nat-lsr-carry-out (rewrite)
  (implies (and (bvp a) (listp a))
           (equal (nth 0 a)
                  (v-alu-nat-lsr-carry-out a)))
  ((enable nth v-to-nat)))


(prove-lemma v-alu-correct-nat-lsr-output (rewrite)
  (implies (bvp a)
           (equal (v-lsr a)
                  (v-alu-nat-lsr-output a)))
  ((enable v-shift-right v-to-nat)))
```

```
(prove-lemma v-alu-correct-nat-lsr (rewrite)
  (implies (bvp a)
           (equal (cvzbv-v-lsr a)
                  (v-alu-nat-lsr a)))
  ((disable v-alu-nat-lsr-carry-out v-alu-nat-lsr-output v-lsr)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                     V-NOT                   ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn v-alu-nat-not-output (a)
  (nat-to-v (sub1 (difference (exp 2 (length a))
                              (v-to-nat a)))
            (length a)))


(defn v-alu-nat-not (a)
  (cvzbv f
         f
         (v-alu-nat-not-output a)))


(prove-lemma v-alu-correct-nat-not-output (rewrite)
  (implies (bvp a)
           (equal (v-not a)
                  (v-alu-nat-not-output a))))


(prove-lemma v-alu-correct-nat-not (rewrite)
  (implies (bvp a)
           (equal (cvzbv-v-not a)
                  (v-alu-nat-not a))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;      CORRECTNESS -- natural number case     ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn v-alu-nat (c a b op)
  (cond ((equal op #v0000) (v-alu-nat-int-buf a))
        ((equal op #v0001) (v-alu-nat-inc a))
        ((equal op #v0010) (v-alu-nat-adder c a b))
        ((equal op #v0011) (v-alu-nat-adder f a b))
        ((equal op #v0101) (v-alu-nat-dec a))
        ((equal op #v0110) (v-alu-nat-subtracter c a b))
        ((equal op #v0111) (v-alu-nat-subtracter f a b))
        ((equal op #v1010) (v-alu-nat-lsr a))
        ((equal op #v1110) (v-alu-nat-not a))
        (t (v-alu c a b op))))
```

```
(prove-lemma v-alu-correct-nat ()
  (implies (bv2p a b)
           (equal (v-alu c a b op)
                  (v-alu-nat c a b op)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable v-alu v-alu-nat bv2p
           v-alu-correct-nat-int-buf v-alu-correct-nat-adder
           v-alu-correct-nat-inc v-alu-correct-nat-subtracter
           v-alu-correct-nat-dec v-alu-correct-nat-lsr
           v-alu-correct-nat-not)
   (disable v-alu-nat-int-buf v-alu-nat-inc v-alu-nat-adder
            v-alu-nat-dec v-alu-nat-subtracter v-alu-nat-lsr
            v-alu-nat-not)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                 *INTEGERS*                ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;; My approach will be to think as little as possible about integers
;; in general, but rather only to bother with two's complement integers,
;; i.e. those that are in range.


(defn int-to-v (i l)
  ;; works even if i is as low as -2**l (inclusive), so that overflowing
  ;; additions should be handled OK
  (if (ilessp i 0)
      (nat-to-v (iplus i (exp 2 l)) l)
    (nat-to-v i l)))


(defn v-to-int (v)
  (if (lessp (v-to-nat v) (exp 2 (sub1 (length v))))
      (v-to-nat v)
    (idifference (v-to-nat v) (exp 2 (length v)))))


(defn integer-in-rangep (i l)
  (and (ileq (ineg (exp 2 (sub1 l))) i)
       (ilessp i (exp 2 (sub1 l)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;       USEFUL LEMMAS for integers case       ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;
;;; lemmas for v-alu-correct-int-adder-overflowp
;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(prove-lemma iplus-plus (rewrite)
  (implies (and (numberp x) (numberp y))
           (equal (iplus x y) (plus x y)))
  ((enable iplus)))


(prove-lemma ilessp-lessp (rewrite)
  (implies (and (numberp x) (numberp y))
           (equal (ilessp x y) (lessp x y)))
  ((enable ilessp)))


(prove-lemma times-2 (rewrite)
  ;; **** probably don't need this
  (equal (times 2 x) (plus x x)))


(prove-lemma b-to-nat-leq (rewrite)
  ;; **** can probably get rid of this if I don't disable b-to-nat in
  ;; proofs below
  (not (lessp 1 (b-to-nat x))))


(prove-lemma lessp-v-to-nat-exp-with-exp-opened (rewrite)
  (implies (and (bvp a) (equal (length a) (length b))
                (not (equal (length b) 0)))
           (lessp (v-to-nat a)
                  (plus (exp 2 (sub1 (length b))) (exp 2 (sub1 (length b))))))
  ((use (lessp-v-to-nat-exp))
   (disable lessp-v-to-nat-exp-rewrite)))


(prove-lemma fix-int-numberp (rewrite)
  (implies (numberp x)
           (equal (fix-int x) x))
  ((enable-theory integer-defns)))

;;;;;;;;;;;;;;;;;;;;;;;;;
;;; lemmas for v-alu-correct-int-adder-output
;;;;;;;;;;;;;;;;;;;;;;;;;


(prove-lemma iplus3-plus-difference (rewrite)
  (implies (and (numberp c0) (numberp a0) (numberp b0) (numberp d)
                (not (lessp (plus c0 a0 b0) d)))
           (equal (iplus c0 (iplus a0 (iplus b0 (ineg d))))
                  (difference (plus c0 a0 b0) d)))
  ((enable-theory integer-defns)))

;;;;;;;;;;;;;;;;;;;;;;;;;
;;; lemmas for v-alu-correct-int-subtracter-overflowp
;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(prove-lemma v-to-int-v-not (rewrite)
  (implies (and (bvp a)
                (not (equal (length a) 0)))
           (equal (v-to-int (v-not a))
                  (iplus -1 (ineg (v-to-int a)))))
  ((enable-theory integer-defns)))


(disable v-alu-correct-nat-not-output)


(prove-lemma v-alu-correct-int-subtracter-overflowp-lemma (rewrite)
  (implies (and (bv2p a b)
                (not (equal (length a) 0)))
           (equal (iplus (b-to-nat (b-not c))
                         (iplus (v-to-int (v-not a))
                                (v-to-int b)))
                  (idifference (v-to-int b)
                               (iplus (v-to-int a) (b-to-nat c)))))
  ((enable-theory integer-defns)
   (disable v-to-int)))

;;;;;;;;;;;;;;;;;;;;;;;;;
;;; lemmas for v-alu-correct-int-asr-output
;;;;;;;;;;;;;;;;;;;;;;;;;


(prove-lemma idifference-difference (rewrite)
  (implies (and (numberp x) (numberp y) (not (lessp x y)))
           (and (equal (iplus x (ineg y))
                       (difference x y))
                (equal (iplus (ineg y) x)
                       (difference x y))))
  ((enable-theory integer-defns)))


(prove-lemma idiv-quotient (rewrite)
  (implies (and (numberp x) (numberp y))
           (equal (idiv x y)
                  (quotient x y)))
  ((enable-theory integer-defns)))


(prove-lemma idiv-ilessp-0 (rewrite)
  (implies (and (numberp y)
                (not (equal y 0)))
           (equal (ilessp (idiv x y) 0)
                  (ilessp x 0)))
  ((enable-theory integer-defns)))
```

```
(prove-lemma quotient-plus-x-x-2-with-extra-arg (rewrite)
  (equal (quotient (plus x x y) 2)
         (plus x (quotient y 2)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable plus-add1)))


(prove-lemma plus-quotient-2-lemma (rewrite)
  (equal (quotient (plus v (times 2 z)) 2)
         (plus z (quotient v 2))))


(prove-lemma plus-quotient-2 (rewrite)
  (implies (not (lessp a0 exp))
           (equal (plus exp
                        (quotient (difference a0 exp) 2))
                  (quotient (plus a0 exp) 2)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable quotient-plus-x-x-2-with-extra-arg
           lessp-remainder remainder-quotient-elim difference-elim
           associativity-of-plus plus-quotient-2-lemma
           commutativity-of-plus commutativity2-of-plus)))


(prove-lemma iplus-div-hack1 (rewrite)
  (implies (and (numberp a0) (numberp exp))
           (equal (iplus (idiv (iplus a0
                                      (ineg exp))
                               2)
                         exp)
                  (quotient (plus a0 exp) 2)))
  ((enable-theory integer-defns)))


(disable quotient-plus-x-x-2-with-extra-arg)

(disable plus-quotient-2-lemma)

(disable plus-quotient-2)


(prove-lemma lessp-quotient-2 (rewrite)
  (iff (lessp (quotient x 2) y)
       (lessp x (plus y y))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                   V-ADDER                   ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defn v-alu-int-adder-output (c a b)
  ;; specification function for v-adder-output
  (int-to-v (iplus (b-to-nat c) (iplus (v-to-int a) (v-to-int b)))
            (length a)))


(defn v-alu-int-adder-overflowp (c a b)
  (not (integer-in-rangep
        (iplus (b-to-nat c) (iplus (v-to-int a) (v-to-int b)))
        (length a))))


(defn v-alu-int-adder (c a b)
  ;; specification function for cvzbv-v-adder
  (cvzbv (v-adder-carry-out         c a b)
         (v-alu-int-adder-overflowp c a b)
         (v-alu-int-adder-output    c a b)))


(prove-lemma v-alu-correct-int-adder-overflowp (rewrite)
  (implies (and (bv2p a b)
                (not (equal (length a) 0)))
           (equal (v-adder-overflowp c a b)
                  (v-alu-int-adder-overflowp c a b)))
  ((enable-theory integer-metas naturals-metas)
   (disable v-adder-output b-to-nat)))


(prove-lemma v-alu-correct-int-adder-output (rewrite)
  (implies (and (bv2p a b)
                (not (equal (length a) 0)))
           (equal (v-adder-output c a b)
                  (v-alu-int-adder-output c a b)))
  ((enable-theory integer-metas naturals-metas)
   (disable v-adder-output b-to-nat)))


(prove-lemma v-alu-correct-int-adder (rewrite)
  (implies (and (bv2p a b)
                (not (equal (length a) 0)))
           (equal (cvzbv-v-adder c a b)
                  (v-alu-int-adder c a b)))
  ((disable v-adder-carry-out v-adder-overflowp
            v-adder-output v-alu-int-adder-overflowp
            v-alu-int-adder-output
            v-alu-correct-nat-adder)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                   V-INC                     ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defn v-alu-int-inc (a)
  ;; specification function for cvzbv-inc -- it seems quite reasonable
  ;; to use the specification function for the adder to express it
  (v-alu-int-adder t a (int-to-v 0 (length a))))


(prove-lemma v-alu-correct-int-inc (rewrite)
  (implies (and (bvp a)
                (not (equal (length a) 0)))
           (equal (cvzbv-inc a)
                  (v-alu-int-inc a)))
  ((disable v-alu-correct-nat-inc v-alu-int-adder cvzbv-v-adder)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                  V-SUBTRACTER                ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn v-alu-int-subtracter-output (c a b)
  ;;   The idea is that it's b - (c + a).
  (int-to-v (idifference (v-to-int b)
                         (iplus (v-to-int a) (b-to-nat c)))
           (length a)))


(defn v-alu-int-subtracter-overflowp (c a b)
  (not (integer-in-rangep
        (idifference (v-to-int b)
                     (iplus (v-to-int a) (b-to-nat c)))
        (length a))))


(defn v-alu-int-subtracter (c a b)
  (cvzbv (v-subtracter-carry-out         c a b)
         (v-alu-int-subtracter-overflowp c a b)
         (v-alu-int-subtracter-output    c a b)))


(prove-lemma v-alu-correct-int-subtracter-overflowp (rewrite)
  (implies (and (bv2p a b)
                (not (equal (length a) 0)))
           (equal (v-subtracter-overflowp c a b)
                  (v-alu-int-subtracter-overflowp c a b)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable bv2p v-alu-correct-int-adder-overflowp
           bvp-v-not length-v-not v-alu-int-adder-overflowp
           v-alu-int-subtracter-overflowp v-subtracter-overflowp
           v-alu-correct-int-subtracter-overflowp-lemma)))
```

```
(prove-lemma v-alu-correct-int-subtracter-output (rewrite)
  (implies (and (bv2p a b)
                (not (equal (length a) 0)))
           (equal (v-subtracter-output c a b)
                  (v-alu-int-subtracter-output c a b)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable v-subtracter-output v-alu-correct-int-adder-output
           v-alu-int-subtracter-output v-alu-int-adder-output
           v-alu-correct-int-subtracter-overflowp-lemma
           length-v-not bv2p bvp-v-not)))


(prove-lemma v-alu-correct-int-subtracter (rewrite)
  (implies (and (bv2p a b)
                (not (equal (length a) 0)))
           (equal (cvzbv-v-subtracter c a b)
                  (v-alu-int-subtracter c a b)))
  ((disable v-subtracter-carry-out v-subtracter-overflowp
            v-subtracter-output v-alu-int-subtracter-overflowp
            v-alu-int-subtracter-output
            v-alu-correct-nat-subtracter-carry-out
            v-alu-correct-nat-subtracter)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                    V-DEC                    ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn v-alu-int-dec (a)
  ;; specification function for cvzbv-dec -- it seems quite reasonable
  ;; to use the specification function for the adder to express it
  (v-alu-int-subtracter t (int-to-v 0 (length a)) a))


(prove-lemma v-alu-correct-int-dec (rewrite)
  (implies (and (bvp a)
                (not (equal (length a) 0)))
           (equal (cvzbv-dec a)
                  (v-alu-int-dec a)))
  ((disable v-alu-int-subtracter cvzbv-v-subtracter
            v-alu-correct-nat-dec v-alu-correct-nat-subtracter)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                    V-ASR                  ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn v-alu-int-asr-output (a)
  (int-to-v (idiv (v-to-int a) 2)
           (length a)))
```

```
(defn v-alu-int-asr (a)
  (cvzbv (v-alu-nat-lsr-carry-out a);; same as lsr case
         f
         (v-alu-int-asr-output a)))


(prove-lemma v-alu-correct-int-asr-output (rewrite)
  (implies (and (bvp a)
                (not (equal (length a) 0)))
           (equal (v-asr a)
                  (v-alu-int-asr-output a)))
  ((enable-theory integer-metas naturals-metas)
   (enable v-shift-right v-to-nat)))


(prove-lemma v-alu-correct-int-asr (rewrite)
  (implies (bvp a)
           (equal (cvzbv-v-asr a)
                  (v-alu-int-asr a)))
  ((disable v-alu-nat-lsr-carry-out v-alu-nat-lsr-output v-lsr)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                    V-NEG                    ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn v-alu-int-neg (a)
  (v-alu-int-subtracter f a (int-to-v 0 (length a))))


(prove-lemma v-alu-correct-int-neg (rewrite)
  (implies (and (bvp a)
                (not (equal (length a) 0)))
           (equal (cvzbv-neg a)
                  (v-alu-int-neg a)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable *1*ilessp cvzbv-neg v-alu-int-neg int-to-v
           v-alu-correct-int-subtracter bv2p bvp-nat-to-v
           length-nat-to-v)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;                      V-NOT                  ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn v-alu-int-not-output (a)
  (v-alu-int-subtracter-output t a (int-to-v 0 (length a))))


(defn v-alu-int-not (a)
  (cvzbv f f (v-alu-int-not-output a)))
```

```
(prove-lemma v-alu-correct-int-not-output (rewrite)
  (implies (and (bvp a)
                (not (equal (length a) 0)))
           (equal (v-not a)
                  (v-alu-int-not-output a)))
  ((disable-theory t)
   (enable-theory ground-zero integer-defns naturals)
   (enable lessp-v-to-nat-exp
           v-to-nat-of-nat-to-v length-nat-to-v
           v-alu-correct-nat-not-output
           v-alu-nat-not-output
           v-alu-int-not-output
           v-alu-int-subtracter-output
           int-to-v v-to-int
           idifference ileq b-to-nat *1*b-to-nat)))


(prove-lemma v-alu-correct-int-not (rewrite)
  (implies (and (bvp a)
                (not (equal (length a) 0)))
           (equal (cvzbv-v-not a)
                  (v-alu-int-not a))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;         CORRECTNESS -- integer case         ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defn v-alu-int (c a b op)
  (cond ((equal op #v0000) (v-alu-nat-int-buf a))
        ((equal op #v0001) (v-alu-int-inc a))
        ((equal op #v0010) (v-alu-int-adder c a b))
        ((equal op #v0011) (v-alu-int-adder f a b))
        ((equal op #v0100) (v-alu-int-neg a))
        ((equal op #v0101) (v-alu-int-dec a))
        ((equal op #v0110) (v-alu-int-subtracter c a b))
        ((equal op #v0111) (v-alu-int-subtracter f a b))
        ((equal op #v1001) (v-alu-int-asr a))
        ((equal op #v1110) (v-alu-int-not a))
        (t (v-alu c a b op))))
```

```
(prove-lemma v-alu-correct-int ()
  (implies (and (bv2p a b)
                (not (equal (length a) 0)))
           (equal (v-alu c a b op)
                  (v-alu-int c a b op)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable v-alu v-alu-int bv2p
           v-alu-correct-nat-int-buf v-alu-correct-int-adder
           v-alu-correct-int-inc v-alu-correct-int-subtracter
           v-alu-correct-int-dec v-alu-correct-int-asr
           v-alu-correct-int-neg v-alu-correct-int-not)
   (disable v-alu-nat-int-buf v-alu-int-inc v-alu-int-adder
            v-alu-int-dec v-alu-int-subtracter v-alu-int-asr
            v-alu-correct-nat)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;               *FINAL DISABLES*              ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(disable v-alu-correct-nat-int-buf)

(disable v-alu-correct-nat-adder-carry-out)

(disable v-alu-correct-nat-adder-output)

(disable v-alu-correct-nat-adder)

(disable v-alu-correct-nat-inc)

(disable v-alu-correct-nat-subtracter-carry-out)

(disable v-alu-correct-nat-subtracter-output)

(disable v-alu-correct-nat-subtracter)

(disable v-alu-correct-nat-dec)

(disable v-alu-correct-nat-lsr-carry-out)

(disable v-alu-correct-nat-lsr-output)

(disable v-alu-correct-nat-lsr)

(disable v-alu-correct-nat-not-output)

(disable v-alu-correct-nat-not)


(disable v-alu-correct-int-adder-overflowp)
```

```
(disable v-alu-correct-int-adder-output)

(disable v-alu-correct-int-adder)

(disable v-alu-correct-int-inc)

(disable v-alu-correct-int-subtracter-overflowp)

(disable v-alu-correct-int-subtracter-output)

(disable v-alu-correct-int-subtracter)

(disable v-alu-correct-int-dec)

(disable v-alu-correct-int-asr-output)

(disable v-alu-correct-int-asr)

(disable v-alu-correct-int-neg)

(disable v-alu-correct-int-not)
```

14.72 "flag-interpretation.events"


```
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    FLAG-INTERPRETATION.EVENTS
;;;
;;;    Bishop Brock
;;;    Computational Logic, Inc.
;;;    May, 1993
;;;
;;;    Matt Kaufmann did much of the work when he proved the ALU interpretation
;;;    lemmas.  The lemmas in this file simply repackage the things he proved
;;;    into a potentially more useful format.  Since I didn't know his state of
;;;    mind concerning the libraries when he did his proofs, mine may have many
;;;    more DISABLE/ENABLE hints than really necessary.
;;;
;;;    The approach taken here is to prove a general lemma about the
;;;    STORE-RESULTP interpretations of the FLAGS, after a particular
;;;    instruction, and a minimal choice of set flags. In other words, assuming
;;;    that the current instruction performs some operation, these lemmas tell
;;;    how subsequent instructions would interpret the flags. E.g.,
;;;
;;;    (IMPLIES
;;;     (AND (BV2P a b)
;;;          <some flags in SET-FLAGS are set to be updated>)
;;;     (EQUAL (STORE-RESULTP (UPDATE-FLAGS flags set-flags <alu-output>)
;;;                           <store-cc code>)
;;;            <some meaningful arithmetic or logical interpretation>)).
;;;
;;;    These lemmas are not stored as rewrite rules because they simply serve
;;;    as consistency checks for the specification.  They would be necessary,
;;;    however, for someone trying to prove an assembly language program on the
;;;    FM9001.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;;  This file was first developed after "flatten.events", which has the
;;  following, seemingly extremely important macro invocation:


(better-disable-back-to xs-suffice-for-reset-lemma)

;;  These are very bothersome lemmas!


(disable v-alu-correct-int-not-output)

(disable v-alu-correct-nat-not-output)

(disable v-alu-correct-nat-lsr-carry-out)
```

```
(disable v-adder-output=v-sum)

(disable v-alu-correct-nat-inc)

(disable v-alu-correct-nat-adder)

(disable v-alu-correct-int-asr)

(disable v-alu-correct-nat-lsr)

;;  These are very useful lemmas to have on.


(enable v-to-nat-of-nat-to-v)

(enable iplus-0-left)

(enable iplus-0-right)

(enable commutativity-of-iplus)

;;
;; These maybe go elsewhere.


(prove-lemma v-negp->v-nzerop (rewrite)
  (implies
   (v-negp v)
   (v-nzerop v))
  ;;Hint
  ((enable v-negp v-nzerop)))


(prove-lemma v-zerop-implies-v-to-nat-0 (rewrite)
  (implies
   (v-zerop a)
   (equal (v-to-nat a) 0))
  ;;Hint
  ((enable v-nzerop v-to-nat)))


(prove-lemma boolp-flag-extracters (rewrite)
  (and
   (boolp (c-flag (update-flags flags set-flags cvzbv)))
   (boolp (v-flag (update-flags flags set-flags cvzbv)))
   (boolp (n-flag (update-flags flags set-flags cvzbv)))
   (boolp (z-flag (update-flags flags set-flags cvzbv))))
  ;;Hint
  ((enable update-flags)
   (enable-theory flags-theory)))
```

```
(prove-lemma v-to-nat-equal (rewrite)
  (implies
   (bv2p a b)
   (equal (equal (v-to-nat a) (v-to-nat b))
          (equal a b)))
  ;;Hint
  ((induct (v-iff a b))
   (enable bvp v-to-nat)))

; Prove that NAT-TO-V on 0 gives a V-ZEROP bvp.


(prove-lemma v-nzerop-nat-to-v-zero (rewrite)
  (implies
   (zerop z)
   (not (v-nzerop (nat-to-v z l))))
  ;;Hint
  ((enable v-nzerop nat-to-v)))

;;  A nice fact about V-NOT.


(prove-lemma v-not-v-not (rewrite)
  (implies
   (bvp a)
   (equal (v-not (v-not a))
          a))
  ;;Hint
  ((enable v-not bvp)))

; Maybe we don't want this one around all the time. Redundant??


(prove-lemma rewrite-equal-v-to-nat-0 (rewrite)
  (equal (equal (v-to-nat a) 0)
         (v-zerop a))
  ;;Hint
  ((enable v-to-nat v-zerop v-nzerop)))


(disable rewrite-equal-v-to-nat-0)

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;  PRELIMINARIES
;;;
;;;  Definition of constants for readability.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


(defn cc-cc () #v0000)

(defn cc-cs () #v0001)
```

```
(defn cc-vc () #v0010)

(defn cc-vs () #v0011)

(defn cc-pl () #v0100)

(defn cc-mi () #v0101)

(defn cc-ne () #v0110)

(defn cc-eq () #v0111)

(defn cc-hi () #v1000)

(defn cc-ls () #v1001)

(defn cc-ge () #v1010)

(defn cc-lt () #v1011)

(defn cc-gt () #v1100)

(defn cc-le () #v1101)

(defn cc-t  () #v1110)

(defn cc-f  () #v1111)


(deftheory condition-code-theory
  (cc-cc cc-cs cc-vc cc-vs cc-pl cc-mi cc-ne cc-eq cc-hi cc-ls cc-ge cc-lt
         cc-gt cc-le cc-t cc-f))


(defn op-move () #v0000)

(defn op-inc  () #v0001)

(defn op-addc () #v0010)

(defn op-add  () #v0011)

(defn op-neg  () #v0100)

(defn op-dec  () #v0101)

(defn op-subb () #v0110)

(defn op-sub  () #v0111)

(defn op-ror  () #v1000)

(defn op-asr  () #v1001)
```

```
(defn op-lsr  () #v1010)

(defn op-xor  () #v1011)

(defn op-or   () #v1100)

(defn op-and  () #v1101)

(defn op-not  () #v1110)

(defn op-m15  () #v1111)


(deftheory opcode-theory
  (op-move op-inc op-addc op-add op-neg op-dec op-subb op-sub op-ror op-asr
           op-lsr op-xor op-or op-and op-not op-m15))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;  BIT VECTORS
;;;
;;;  Here we include interpretations valid for all functions of the ALU, as
;;;  well as some interpretation lemmas for the carry out on logical shifts.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

;;;  First, prove that the Z bit and N bit are always set as expected.

; Go get a cup of coffee while this one grinds out!

(prove-lemma zb-v-alu ()
  (equal (zb (v-alu c a b op-code))
         (v-zerop (bv (v-alu c a b op-code))))
  ((enable zb v-alu bv)))


(prove-lemma n-v-alu ()
  (equal (n (v-alu c a b op-code))
         (v-negp (bv (v-alu c a b op-code))))
  ((enable n v-alu bv)))
```

```
(prove-lemma universal-flags-interpretations ()
  (let
    ((new-flags (update-flags flags set-flags (v-alu c a b op-code))))
    (implies
     (and (bvp a)
          (boolp c)
          (not (zerop (length a))))
     (and
      (implies
       (z-set set-flags)
       (and (equal (store-resultp (cc-ne) new-flags)
                   (v-nzerop (bv (v-alu c a b op-code))))
            (equal (store-resultp (cc-eq) new-flags)
                   (not (v-nzerop (bv (v-alu c a b op-code)))))))
      (implies
       (n-set set-flags)
       (and (equal (store-resultp (cc-pl) new-flags)
                   (not (v-negp (bv (v-alu c a b op-code)))))
            (equal (store-resultp (cc-mi) new-flags)
                   (v-negp (bv (v-alu c a b op-code))))))
      (equal (store-resultp (cc-t) new-flags)
             t)
      (equal (store-resultp (cc-f) new-flags)
             f))))
  ;;Hint
  ((use (zb-v-alu) (n-v-alu))
   (enable set-flags update-flags store-resultp boolp)
   (enable-theory set-flags-theory flags-theory)))

;;
;;   SUB
;;
;;   The proof that SUB is a compare for bit vectors.
;;
;;   We show that if the result of a subtract is 0, then a=b.
;;   We first prove a lemma about V-SUM.  This is a V-SUM lemma because
;;   V-SUBTRACTER-OUTPUT is defined in terms of V-ADDER-OUTPUT which can be
;;   rewritten to V-SUM.


(prove-lemma v-sum-on-not-a-a (rewrite)
  (implies
   (bv2p a b)
   (equal (v-nzerop (v-sum t a b))
          (not (equal (v-not a) b))))
  ;;Hint
  ((enable v-nzerop v-sum v-not)
   (disable v-alu-correct-nat-not-output)))
```

```
(prove-lemma if-v-subtracter-output=0-then-a=b (rewrite)
  (implies
   (bv2p a b)
   (equal (v-nzerop (v-subtracter-output f a b))
          (not (equal a b))))
  ;;Hint
  ((disable v-alu-correct-nat-not-output)
   (enable v-subtracter-output v-adder-output=v-sum)))

;; Good for all subtracts!


(prove-lemma zero-flag-interpretation-for-sub ()
  (let
    ((sub-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-sub)))))
     (implies
      (and (bv2p a b)
           (z-set set-flags))
      (equal (z-flag sub-flags)
             (equal a b))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu v-alu-nat-inc
           v-alu-nat-adder v-alu-nat-adder-carry-out
           v-alu-nat-dec v-alu-nat-subtracter v-alu-nat-subtracter-carry-out
           v-alu-nat-subtracter-output
           c v n cv bv *1*exp *1*b-to-nat
           zb)
   (enable-theory set-flags-theory flags-theory)
   (disable v-subtracter-output v-alu-correct-nat-subtracter
            v-alu-correct-nat-subtracter-output)))


(prove-lemma flags-interpretation-sub ()
  (let
    ((new-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-sub)))))
    (implies
     (and (bv2p a b)
          (z-set set-flags))
     (and (equal (store-resultp (cc-ne) new-flags)
                 (not (equal a b)))
          (equal (store-resultp (cc-eq) new-flags)
                 (equal a b)))))
  ;;Hint
  ((use (zero-flag-interpretation-for-sub))
   (enable set-flags store-resultp)
   (disable v-alu-correct-nat-inc v-alu-correct-nat-adder)))

;;
;;  ROR
;;
;;  Carry out of ROR
```

```
(prove-lemma v-negp-v-shift-right (rewrite)
  (implies
   (and (not (equal (length a) 0))
        (boolp c))
   (equal (v-negp (v-shift-right a c))
          c))
  ;;Hint
  ((enable v-negp v-shift-right append v-buf)
   (induct (length a))))


(prove-lemma flags-interpretation-ror ()
  (let
    ((new-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-ror)))))
    (implies
     (and (bvp a)
          (boolp (c-flag flags))
          (not (zerop (length a))))
     (and
      (implies
       (c-set set-flags)
       (and (equal (store-resultp (cc-cc) new-flags)
                   (not (nth 0 a)))
            (equal (store-resultp (cc-cs) new-flags)
                   (nth 0 a))))
      (implies
       (n-set set-flags)
       (and (equal (store-resultp (cc-pl) new-flags)
                   (not (c-flag flags)))
            (equal (store-resultp (cc-mi) new-flags)
                   (c-flag flags)))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu c n bv)
   (enable-theory set-flags-theory flags-theory)))

;;
;;  LSR
;;
```

```
(prove-lemma flags-interpretation-lsr ()
  (let
    ((new-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-lsr)))))
    (implies
     (and (bvp a)
          (not (zerop (length a)))
          (c-set set-flags))
     (and (equal (store-resultp (cc-cc) new-flags)
                 (not (nth 0 a)))
          (equal (store-resultp (cc-cs) new-flags)
                 (nth 0 a)))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu c)
   (enable-theory set-flags-theory flags-theory)))

;;
;;   XOR
;;
;;   Prove that XOR can also be used as a compare.


(prove-lemma flags-interpretation-xor ()
  (let
    ((new-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-xor)))))
    (implies
     (and (bv2p a b)
          (not (zerop (length a)))
          (z-set set-flags))
     (and (equal (store-resultp (cc-ne) new-flags)
                 (not (equal a b)))
          (equal (store-resultp (cc-eq) new-flags)
                 (equal a b)))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu zb)
   (enable-theory set-flags-theory flags-theory)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    NATURALS
;;;
;;;    Interpretation of the flags for the cases where the arguments are
;;;    thought of as natural numbers.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

;;
;;  MOV
;;
;;  Zero clear and set on natural moves.
```

```
(prove-lemma flags-interpretation-nat-move ()
  (let
    ((move-flags (update-flags flags set-flags
                               (v-alu (c-flag flags) a b (op-move))))
     (m15-flags (update-flags flags set-flags
                               (v-alu (c-flag flags) a b (op-m15)))))
    (implies
     (and (bv2p a b)
          (z-set set-flags))
     (and
      (equal (store-resultp (cc-ne) move-flags)
             (not (equal (v-to-nat a) 0)))
      (equal (store-resultp (cc-eq) move-flags)
             (equal (v-to-nat a) 0))
      (equal (store-resultp (cc-ne) m15-flags)
             (not (equal (v-to-nat a) 0)))
      (equal (store-resultp (cc-eq) m15-flags)
             (equal (v-to-nat a) 0)))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv *1*exp *1*b-to-nat zb
           rewrite-equal-v-to-nat-0)
   (enable-theory set-flags-theory flags-theory)))

;;
;;  INC
;;
;;  Carry clear and set on natural increment.


(prove-lemma flags-interpretation-nat-inc ()
  (let
    ((inc-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-inc)))))
    (implies
     (and (bv2p a b)
          (c-set set-flags))
     (and
      (equal (store-resultp (cc-cc) inc-flags)
             (lessp (add1 (v-to-nat a)) (exp 2 (length a))))
      (equal (store-resultp (cc-cs) inc-flags)
             (not (lessp (add1 (v-to-nat a)) (exp 2 (length a))))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu v-alu-nat-inc
           v-alu-nat-adder v-alu-nat-adder-carry-out
           c v n cv bv *1*exp *1*b-to-nat)
   (enable-theory set-flags-theory flags-theory)))

;;
;;  ADDC
;;
;;  Carry clear and set on natural addition with carry.
```

```
(prove-lemma flags-interpretation-nat-addc ()
  (let
    ((add-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-addc)))))
    (implies
     (and (bv2p a b)
          (c-set set-flags))
     (and
      (equal (store-resultp (cc-cc) add-flags)
             (lessp (plus (b-to-nat (c-flag flags))
                          (v-to-nat a)
                          (v-to-nat b)) (exp 2 (length a))))
      (equal (store-resultp (cc-cs) add-flags)
             (not (lessp (plus (b-to-nat (c-flag flags))
                               (v-to-nat a)
                               (v-to-nat b))
                         (exp 2 (length a))))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu v-alu-nat-inc
           v-alu-nat-adder v-alu-nat-adder-carry-out
           c v n cv bv *1*exp *1*b-to-nat)
   (enable-theory set-flags-theory flags-theory)))

;;
;;  ADD
;;
;;  Carry clear and set on natural addition.
```

```
(prove-lemma flags-interpretation-nat-add ()
  (let
    ((add-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-add)))))
    (implies
     (and (bv2p a b)
          (c-set set-flags))
     (and
      (equal (store-resultp (cc-cc) add-flags)
             (lessp (plus (v-to-nat a) (v-to-nat b)) (exp 2 (length a))))
      (equal (store-resultp (cc-cs) add-flags)
             (not (lessp (plus (v-to-nat a) (v-to-nat b))
                         (exp 2 (length a))))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu v-alu-nat-inc
           v-alu-nat-adder v-alu-nat-adder-carry-out
           c v n cv bv *1*exp *1*b-to-nat)
   (enable-theory set-flags-theory flags-theory)))

;;
;;  DEC
;;
;;  Carry clear and set on natural decrement.
;;  Zero clear and set on natural decrement.
;;  HIGHER and LOWER OR SAME on natural decrement.

;; Hats off to Warren for this record-breaking name!

(prove-lemma when-the-mantissa-isnt-0-then-neither-is-the-exponential (rewrite)
  (implies
   (not (zerop mantissa))
   (not (equal (exp mantissa exponent) 0)))
  ;;hint
  ((enable exp)))


(defn double-difference-induction (n m l)
  (if (zerop n)
      t
    (if (zerop m)
        t
      (if (zerop l)
          t
        (double-difference-induction (difference n l) (difference m l) l)))))
```

```
(prove-lemma quotient-lessp-double-remainder$help ()
  (implies
   (and (lessp n m)
        (not (zerop n))
        (not (zerop m))
        (not (zerop l))
        (equal (remainder n l) 0)
        (equal (remainder m l) 0))
   (lessp (quotient n l) (quotient m l)))
  ;;Hint
  ((induct (double-difference-induction n m l))))


(prove-lemma v-nzerop-nat-to-v$help (rewrite)
  (implies
   (and (not (zerop l))
        (not (zerop n))
        (equal (remainder n 2) 0)
        (lessp n (exp 2 l)))
   (lessp (quotient n 2)
          (exp 2 (sub1 l))))
  ;;Hint
  ((enable remainder-exp quotient-exp)
   (use (quotient-lessp-double-remainder$help
        (n n) (m (exp 2 l)) (l 2)))))


(disable v-nzerop-nat-to-v$help)


(prove-lemma v-nzerop-nat-to-v (rewrite)
  (implies
   (lessp n (exp 2 l))
   (equal (v-nzerop (nat-to-v n l))
          (not (zerop n))))
  ;;Hint
  ((enable v-nzerop nat-to-v quotient exp
           v-nzerop-nat-to-v$help)
   (induct (nat-to-v n l))))


(prove-lemma lessp-sub1-plus-a-b-b (rewrite)
  (equal (lessp (sub1 (plus a b)) b)
         (and (zerop a)
              (not (zerop b)))))
```

```
(prove-lemma lessp-v-to-nat-exp-2-length-a (rewrite)
  (implies
   (equal l (length a))
   (equal (lessp (v-to-nat a) (exp 2 l))
          t))
  ;;Hint
  ((enable v-to-nat exp length)))


(prove-lemma lessp-v-to-nat-exp-2-length-a$linear (rewrite)
  (lessp (v-to-nat a) (exp 2 (length a))))


(prove-lemma flags-interpretation-nat-dec ()
  (let
    ((dec-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-dec)))))
    (implies
     (bv2p a b)
     (and
      (implies
       (c-set set-flags)
       (and
        (equal (store-resultp (cc-cc) dec-flags)
               (not (equal (v-to-nat a) 0)))
        (equal (store-resultp (cc-cs) dec-flags)
               (equal (v-to-nat a) 0))))
      (implies
       (and (z-set set-flags)
            (not (equal (length a) 0)))
       (and
        (equal (store-resultp (cc-ne) dec-flags)
               (not (equal (v-to-nat a) 1)))
        (equal (store-resultp (cc-eq) dec-flags)
               (equal (v-to-nat a) 1)))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu v-alu-nat-inc
           ;;v-alu-nat-adder v-alu-nat-adder-carry-out
           v-alu-nat-dec v-alu-nat-subtracter v-alu-nat-subtracter-carry-out
           v-alu-nat-subtracter-output
           c v n cv bv zb *1*exp *1*b-to-nat
           equal-sub1-0 equal-exp-1)
   (expand (remainder (sub1 (plus (v-to-nat a)
                                  (exp 2 (length a))))
                      (exp 2 (length a))))
   (enable-theory set-flags-theory flags-theory)))

;;
;;  SUBB
;;
;;  Carry clear and set, and HIGHER or LOWER OR SAME.  The lemmas don't work as
;;  well when proved together.
```

```
(prove-lemma remainder-theorem-for-subb (rewrite)
  (implies
   (and (lessp a b)
        (lessp b c))
   (equal (remainder (difference (plus b c) a) c)
          (difference b a)))
  ;;Hint
  ((expand (remainder (difference (plus b c) a) c))))

;;  Move elsewhere!

(prove-lemma difference-linear (rewrite)
  (implies
   (lessp a b)
   (equal (lessp (difference a c) b)
          t)))

;;  Carry clear and set on natural subb.


(prove-lemma flags-interpretation-nat-subb-carry ()
  (let
    ((subb-flags (update-flags flags set-flags
                               (v-alu (c-flag flags) a b (op-subb)))))
    (implies
     (and (bv2p a b)
          (c-set set-flags))
       (and
        (equal (store-resultp (cc-cc) subb-flags)
               (not (lessp (v-to-nat b)
                           (plus (v-to-nat a) (b-to-nat (c-flag flags))))))
        (equal (store-resultp (cc-cs) subb-flags)
               (lessp (v-to-nat b)
                      (plus (v-to-nat a) (b-to-nat (c-flag flags))))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu v-alu-nat-inc
          v-alu-nat-adder v-alu-nat-adder-carry-out
          v-alu-nat-dec v-alu-nat-subtracter v-alu-nat-subtracter-carry-out
          c v n cv bv *1*exp *1*b-to-nat)
   (enable-theory set-flags-theory flags-theory)))

;;  Carry and Zero on natural SUBB.
```

```
(prove-lemma flags-interpretation-nat-subb-higher ()
  (let
    ((subb-flags (update-flags flags set-flags
                               (v-alu (c-flag flags) a b (op-subb)))))
    (implies
     (and (bv2p a b)
          (c-set set-flags)
          (z-set set-flags))
     (and
      (equal (store-resultp (cc-hi) subb-flags)
             (lessp (plus (v-to-nat a) (b-to-nat (c-flag flags)))
                    (v-to-nat b)))
      (equal (store-resultp (cc-ls) subb-flags)
             (not (lessp (plus (v-to-nat a) (b-to-nat (c-flag flags)))
                         (v-to-nat b)))))))
  ((enable set-flags update-flags store-resultp v-alu v-alu-nat-inc
           v-alu-nat-adder v-alu-nat-adder-carry-out
           v-alu-nat-dec v-alu-nat-subtracter v-alu-nat-subtracter-carry-out
           v-alu-nat-subtracter-output
           c v n cv bv zb *1*exp)
   (enable-theory set-flags-theory flags-theory)))

;;
;;  SUB
;;


(prove-lemma carry-flag-nat-interpretation-for-sub ()
  (let
    ((sub-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-sub)))))
    (implies
     (and (bv2p a b)
          (c-set set-flags))
     (equal (c-flag sub-flags)
            (lessp (v-to-nat b) (v-to-nat a)))))

  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu v-alu-nat-inc
           v-alu-nat-adder v-alu-nat-adder-carry-out
           v-alu-nat-dec v-alu-nat-subtracter v-alu-nat-subtracter-carry-out
           v-alu-nat-subtracter-output
           c v n cv bv *1*exp *1*b-to-nat
           zb)
   (enable-theory set-flags-theory flags-theory)))

;;  The flags interpretation lemma for natural subtract.
```

```
(prove-lemma flags-interpretation-nat-sub ()
  (let
   ((sub-flags (update-flags flags set-flags
                             (v-alu (c-flag flags) a b (op-sub)))))
    (implies
     (bv2p a b)
     (and
      ;; Carry clear and set.
      (implies
       (c-set set-flags)
       (and (equal (store-resultp (cc-cc) sub-flags)
                   (not (lessp (v-to-nat b) (v-to-nat a))))
            (equal (store-resultp (cc-cs) sub-flags)
                   (lessp (v-to-nat b) (v-to-nat a)))))
      ;;  Unsigned comparisons
      (implies
       (and (c-set set-flags)
            (z-set set-flags))
       (and (equal (store-resultp (cc-hi) sub-flags)
                   (lessp (v-to-nat a) (v-to-nat b)))
            (equal (store-resultp (cc-ls) sub-flags)
                   (not (lessp (v-to-nat a) (v-to-nat b)))))))))
  ;;Hint
  ((use (zero-flag-interpretation-for-sub)
        (carry-flag-nat-interpretation-for-sub))
   (enable c-set z-set store-resultp)
   (disable boolp)))

;;
;;  LSR
;;

;;  Carry clear and set for natural shift right.
```

```
(prove-lemma flags-interpretation-nat-lsr-carry ()
  (let
    ((lsr-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-lsr)))))
    (implies
     (and (bv2p a b)
          (not (zerop (length a)))
          (c-set set-flags))
     (and
      (equal (store-resultp (cc-cc) lsr-flags)
             (equal (remainder (v-to-nat a) 2) 0))
      (equal (store-resultp (cc-cs) lsr-flags)
             (not (equal (remainder (v-to-nat a) 2) 0))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu v-alu-nat-inc
           v-alu-nat-lsr v-alu-nat-lsr-carry-out
           v-alu-correct-nat-lsr-carry-out
           c v n cv bv *1*exp *1*b-to-nat)
   (enable-theory set-flags-theory flags-theory)
   (disable v-negp-v-shift-right open-nth)))

;;  Zero clear and set for natural shift right.
```

```
(prove-lemma flags-interpretation-nat-lsr-zero ()
  (let
    ((lsr-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-lsr)))))
    (implies
     (and (bv2p a b)
          (not (zerop (length a)))
          (z-set set-flags))
     (and
      (equal (store-resultp (cc-ne) lsr-flags)
             (not (equal (quotient (v-to-nat a) 2) 0)))
      (equal (store-resultp (cc-eq) lsr-flags)
             (equal (quotient (v-to-nat a) 2) 0)))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu v-alu-nat-inc
           v-alu-nat-lsr-output
           zb c v n cv bv *1*exp *1*b-to-nat)
   (enable-theory set-flags-theory flags-theory)))

;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;;;
;;;    INTEGERS
;;;
;;;    Interpretation of the flags for the cases where the arguments are
;;;    thought of as integers.
;;;
;;;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

;;
;;  We begin with a raft of useful lemmas.
;;


(prove-lemma idifference-0 (rewrite)
  (implies
   (and (integerp a)
        (integerp b))
   (equal (equal (idifference a b) 0)
          (equal a b)))
  ;;Hint
  ((enable integerp fix-int idifference iplus ineg)))


(prove-lemma v-to-nat-lessp-exp-2-length (rewrite)
  (lessp (v-to-nat a) (exp 2 (length a)))
  ;;Hint
  ((enable v-to-nat exp length)))
```

```
(prove-lemma integerp-v-to-int (rewrite)
  (integerp (v-to-int a))
  ;;Hint
  ((enable integerp v-to-int idifference iplus)))

; Unfortunate, but V-NEGP doesn't BOOLFIX the sign bit, so we can't get a
; hypothesis-free equality.

(prove-lemma v-negp-as-bounds (rewrite)
  (implies
   (bvp x)
   (equal (v-negp x) (not (lessp (v-to-nat x) (exp 2 (sub1 (length x)))))))
  ;;Hint
  ((enable v-to-nat v-negp exp length bvp)))


(disable v-negp-as-bounds)


(prove-lemma negativep-idifference-on-numberps (rewrite)
  (implies
   (and (numberp a)
        (numberp b))
   (equal (negativep (idifference a b))
          (lessp a b)))
  ;;Hint
  ((enable idifference iplus ineg)))


(prove-lemma v-negp->negativep-v-to-int (rewrite)
  (implies
   (bvp a)
   (equal (v-negp a) (negativep (v-to-int a))))
  ;;Hint
  ((enable v-negp-as-bounds v-to-int)))


(disable v-negp->negativep-v-to-int)


(prove-lemma rewrite-equal-v-to-int-0 (rewrite)
  (equal (equal (v-to-int a) 0)
         (v-zerop a))
  ;;Hint
  ((enable integerp v-to-int v-zerop v-nzerop rewrite-equal-v-to-nat-0)))


(disable rewrite-equal-v-to-int-0)
```

```
(prove-lemma int-to-v-v-to-int-0 (rewrite)
  (equal (v-to-int (int-to-v 0 l)) 0)
  ;;Hint
  ((enable v-to-int int-to-v ilessp)))


(prove-lemma bvp-int-to-v (rewrite)
  (bvp (int-to-v i l))
  ;;Hint
  ((enable bvp int-to-v)))


(prove-lemma v-to-int-nat-to-v-0 (rewrite)
  (equal (v-to-int (nat-to-v 0 l))
         0)
  ;;Hint
  ((enable v-to-int nat-to-v)))


(prove-lemma integer-in-rangep-minus-0 (rewrite)
  (integer-in-rangep (minus 0) l)
  ;;Hint
  ((enable integer-in-rangep *1*exp ileq ilessp ineg)))


(prove-lemma not-equal-iplus-minus-0 (rewrite)
  (not (equal (iplus a b) (minus 0)))
  ;;Hint
  ((enable-theory all-integer-defns)))


(prove-lemma lessp-1-plus-x-x (rewrite)
  (equal (lessp 1 (plus x x))
         (not (zerop x))))


(prove-lemma integer-in-rangep-1 (rewrite)
  (implies
   (and (not (zerop l))
        (not (equal l 1)))
   (integer-in-rangep 1 l))
  ;;Hint
  ((enable integer-in-rangep ileq ilessp iplus ineg)
   (expand (exp 2 (sub1 l)))))


(prove-lemma lessp-a-plus-a-a (rewrite)
  (equal (lessp a (plus a a))
         (not (zerop a))))
```

```
(prove-lemma exp-linear-bounds (rewrite)
  (implies
   (and (leq (exp 2 (sub1 n)) v)
        (lessp v (exp 2 n)))
   (not (lessp (exp 2 (sub1 n)) (difference (exp 2 n) v))))
  ;;Hint
  ((expand (exp 2 n))))


(prove-lemma integer-in-rangep-v-to-int (rewrite)
  (implies
   (equal l (length a))
   (integer-in-rangep (v-to-int a) l))
  ;;Hint
  ((enable integer-in-rangep v-to-int)
   (enable-theory all-integer-defns)
   (disable bvp)))

;;  These lemmas are very important for the interpretation of the carry and
;;  negative flags, and not at all straightforward to prove.


(prove-lemma addition-cases-for-integer-in-rangep-lemmas ()
  (implies
   (and (integer-in-rangep a l)
        (integer-in-rangep b l)
        (not (zerop l)))
   (lessp (iplus a b) (exp 2 l)))
  ;;Hint
  ((enable exp integer-in-rangep ilessp ileq ineg iplus int-to-v
           v-negp-as-bounds)))


(prove-lemma subtraction-cases-for-integer-in-rangep-lemmas ()
  (implies
   (and (integer-in-rangep a l)
        (integer-in-rangep b l)
        (not (zerop l)))
   (lessp (iplus b (ineg a)) (exp 2 l)))
  ;;Hint
  ((enable exp integer-in-rangep ilessp ileq ineg iplus int-to-v fix-int
           v-negp-as-bounds)))


(prove-lemma exp-2-l-1<=exp-2-l ()
  (implies
   (not (zerop l))
   (lessp (exp 2 (sub1 l)) (exp 2 l)))
  ;;Hint
  ((enable exp)))
```

```
(prove-lemma lessp-remainder-theorem ()
  (implies
   (lessp n m)
   (lessp (remainder n r) m)))


(prove-lemma integer-in-rangep-the-obvious-way (rewrite)
  (implies
   (and
    (not (zerop l))
    (integerp i)
    (integer-in-rangep i l))
   (equal (v-negp (int-to-v i l))
          (negativep i)))
  ;;Hint
  ((use (lessp-remainder-theorem
         (n i) (m (exp 2 (sub1 l)))
         (r (plus (exp 2 (sub1 l)) (exp 2 (sub1 l))))))
   (enable exp integer-in-rangep v-negp-as-bounds int-to-v)
   (enable-theory all-integer-defns)))


(disable integer-in-rangep-the-obvious-way)


(prove-lemma integer-in-rangep-iplus (rewrite)
  (implies
   (and (integer-in-rangep a l)
        (integer-in-rangep b l)
        (not (zerop l)))
   (equal (integer-in-rangep (iplus a b) l)
          (equal (v-negp (int-to-v (iplus a b) l))
                 (negativep (iplus a b)))))
  ;;Hint
  ((use
    (addition-cases-for-integer-in-rangep-lemmas)
    (exp-2-l-1<=exp-2-l))
   (enable integer-in-rangep exp v-negp-as-bounds int-to-v)
   (enable-theory all-integer-defns)))


(disable integer-in-rangep-iplus)
```

```
(prove-lemma integer-in-rangep-iplus$commuted ()
  (implies
   (and (integer-in-rangep a l)
        (integer-in-rangep b l)
        (not (zerop l)))
   (equal (negativep (iplus a b))
          (iff (integer-in-rangep (iplus a b) l)
               (v-negp (int-to-v (iplus a b) l)))))
  ;;Hint
  ((use (integer-in-rangep-iplus))))


(prove-lemma integer-in-rangep-iplus-carry (rewrite)
  (implies
   (and (integer-in-rangep a l)
        (integer-in-rangep b l)
        (not (zerop l)))
   (equal (integer-in-rangep (iplus 1 (iplus a b)) l)
          (equal (v-negp (int-to-v (iplus 1 (iplus a b)) l))
                 (negativep (iplus 1 (iplus a b))))))
  ;;Hint
  ((use (addition-cases-for-integer-in-rangep-lemmas)
        (exp-2-l-1<=exp-2-l))
   (enable exp integer-in-rangep v-negp-as-bounds int-to-v)
   (enable-theory all-integer-defns)))


(disable integer-in-rangep-iplus-carry)


(prove-lemma integer-in-rangep-iplus-carry$commuted ()
  (implies
   (and (integer-in-rangep a l)
        (integer-in-rangep b l)
        (not (zerop l)))
   (equal (negativep (iplus 1 (iplus a b)))
          (iff (integer-in-rangep (iplus 1 (iplus a b)) l)
               (v-negp (int-to-v (iplus 1 (iplus a b)) l)))))
  ;;Hint
  ((use (integer-in-rangep-iplus-carry))))


(prove-lemma not-integerp-minus-zero (rewrite)
  (equal (integerp (minus x))
         (not (zerop x)))
  ;;Hint
  ((enable integerp)))
```

```
(prove-lemma integer-in-rangep-iplus-ineg (rewrite)
  (implies
   (and (integer-in-rangep a l)
        (integer-in-rangep b l)
        (not (zerop l)))
   (equal (integer-in-rangep (iplus b (ineg a)) l)
          (equal (v-negp (int-to-v (iplus b (ineg a)) l))
                 (negativep (iplus b (ineg a))))))
  ;;Hint
  ((use (subtraction-cases-for-integer-in-rangep-lemmas)
        (exp-2-l-1<=exp-2-l))
   (enable exp integer-in-rangep ilessp ileq ineg iplus int-to-v
           fix-int v-negp-as-bounds)))


(disable integer-in-rangep-iplus-ineg)


(prove-lemma integer-in-rangep-iplus-ineg$commuted ()
  (implies
   (and (integer-in-rangep a l)
        (integer-in-rangep b l)
        (not (zerop l)))
   (equal (negativep (iplus b (ineg a)))
          (iff (integer-in-rangep (iplus b (ineg a)) l)
               (v-negp (int-to-v (iplus b (ineg a)) l)))))
  ;;Hint
  ((use (integer-in-rangep-iplus-ineg))))


(prove-lemma integer-in-rangep-iplus-ineg-carry (rewrite)
  (implies
   (and (integer-in-rangep a l)
        (integer-in-rangep b l)
        (not (zerop l))
        (not (equal l 1)))
   (equal (integer-in-rangep (iplus b (ineg (iplus 1 a))) l)
          (equal (v-negp (int-to-v (iplus b (ineg (iplus 1 a))) l))
                 (negativep (iplus b (ineg (iplus 1 a)))))))
  ;;Hint
  ((use (subtraction-cases-for-integer-in-rangep-lemmas)
        (exp-2-l-1<=exp-2-l))
   (expand (remainder b (plus (exp 2 (sub1 l)) (exp 2 (sub1 l)))))
   (enable exp integer-in-rangep ilessp ileq ineg iplus int-to-v
           idifference fix-int v-negp-as-bounds)))


(disable integer-in-rangep-iplus-ineg-carry)
```

```
(prove-lemma integer-in-rangep-iplus-ineg-carry$commuted ()
  (implies
   (and (integer-in-rangep a l)
        (integer-in-rangep b l)
        (not (zerop l))
        (not (equal l 1)))
   (equal (negativep (iplus b (ineg (iplus 1 a))))
          (iff (v-negp (int-to-v (iplus b (ineg (iplus 1 a))) l))
               (integer-in-rangep (iplus b (ineg (iplus 1 a))) l))))
  ;;Hint
  ((use (integer-in-rangep-iplus-ineg-carry))))


(prove-lemma lessp-1-exp-2 (rewrite)
  (implies
   (not (zerop l))
   (lessp 1 (exp 2 l)))
  ;;Hint
  ((enable exp)))


(prove-lemma v-nzerop-int-to-v (rewrite)
  (implies
   (and (integer-in-rangep n l)
        (not (zerop l)))
   (equal (v-nzerop (int-to-v n l))
          (not (izerop n))))
  ;;Hint
  ((enable int-to-v integer-in-rangep exp)
   (enable-theory all-integer-defns)))


(prove-lemma izerop-inc-v-nzerop (rewrite)
  (implies
   (and (integer-in-rangep a l)
        (not (zerop l)))
   (equal (v-nzerop (int-to-v (iplus 1 a) l))
          (not (izerop (iplus 1 a)))))
  ;;Hint
  ((enable-theory all-integer-defns)
   (enable exp int-to-v integer-in-rangep)))


(prove-lemma v-nzerop-int-to-v-0 (rewrite)
  (implies
   (izerop z)
   (not (v-nzerop (int-to-v z l))))
  ;;Hint
  ((enable int-to-v)
   (enable-theory all-integer-defns)))
```

```
(prove-lemma integer-in-rangep-0 (rewrite)
  (implies
   (izerop a)
   (integer-in-rangep a l))
  ;;Hint
  ((enable-theory all-integer-defns)
   (enable integer-in-rangep)))


(prove-lemma not-v-negp-nat-to-v-0 (rewrite)
  (implies
   (zerop a)
   (not (v-negp (nat-to-v a l))))
  ;;Hint
  ((enable v-negp nat-to-v)))


(prove-lemma length-int-to-v (rewrite)
  (equal (length (int-to-v i l))
         (fix l))
  ;;Hint
  ((enable int-to-v)))


(prove-lemma integer-in-rangep--1 (rewrite)
  (implies
   (not (zerop l))
   (integer-in-rangep -1 l))
  ;;Hint
  ((enable integer-in-rangep)
   (enable-theory all-integer-defns)))

;;
;;  BEGIN LEMMAS
;;

;;
;;
;;  MOVE
;;
;;  Overflow clear and set, Zero clear and set, Negative clear and set, and
;;  inequalities on integer moves.
```

```
(prove-lemma flags-interpretation-int-move ()
  (let
    ((move-flags (update-flags flags set-flags
                               (v-alu (c-flag flags) a b (op-move))))
     (m15-flags (update-flags flags set-flags
                               (v-alu (c-flag flags) a b (op-m15)))))
    (implies
     (bvp a)
     (and
      ;;  No overflow on move.
      (implies
       (v-set set-flags)
       (and (equal (store-resultp (cc-vc) move-flags)
                   (integer-in-rangep (v-to-int a) (length a)))
            (equal (store-resultp (cc-vc) m15-flags)
                   (integer-in-rangep (v-to-int a) (length a)))
            (equal (store-resultp (cc-vs) move-flags)
                   (not (integer-in-rangep (v-to-int a) (length a))))
            (equal (store-resultp (cc-vs) m15-flags)
                   (not (integer-in-rangep (v-to-int a) (length a))))))
      ;;  Zero
      (implies
       (z-set set-flags)
       (and
        (equal (store-resultp (cc-ne) move-flags)
               (not (equal (v-to-int a) 0)))
        (equal (store-resultp (cc-eq) move-flags)
               (equal (v-to-int a) 0))
        (equal (store-resultp (cc-ne) m15-flags)
               (not (equal (v-to-int a) 0)))
        (equal (store-resultp (cc-eq) m15-flags)
               (izerop (v-to-int a)))))
      ;;  Simple negative.
      (implies
       (n-set set-flags)
       (and
        (equal (store-resultp (cc-pl) move-flags)
               (not (negativep (v-to-int a))))
        (equal (store-resultp (cc-mi) move-flags)
               (negativep (v-to-int a)))
        (equal (store-resultp (cc-pl) m15-flags)
               (not (negativep (v-to-int a))))
        (equal (store-resultp (cc-mi) m15-flags)
               (negativep (v-to-int a)))))
      ;;  Inequalities. Note that V = F on a MOV.
      (implies
       (and (v-set set-flags)
            (n-set set-flags))
       (and
        (equal (store-resultp (cc-ge) move-flags)
               (not (negativep (v-to-int a))))
        (equal (store-resultp (cc-lt) move-flags)
               (negativep (v-to-int a)))
        (equal (store-resultp (cc-ge) m15-flags)
```

```
              (not (negativep (v-to-int a))))
       (equal (store-resultp (cc-lt) m15-flags)
              (negativep (v-to-int a)))))
     ;;  Other inequalities. Note that V = F on a MOV.
     (implies
      (and (v-set set-flags)
           (n-set set-flags)
           (z-set set-flags))
      (and
       (equal (store-resultp (cc-gt) move-flags)
              (ilessp 0 (v-to-int a)))
       (equal (store-resultp (cc-le) move-flags)
              (ileq (v-to-int a) 0))
       (equal (store-resultp (cc-gt) m15-flags)
              (ilessp 0 (v-to-int a)))
       (equal (store-resultp (cc-le) m15-flags)
              (ileq (v-to-int a) 0)))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv *1*exp *1*b-to-nat zb
           izerop fix-int ilessp ileq
           v-negp->negativep-v-to-int rewrite-equal-v-to-int-0)
   (enable-theory set-flags-theory flags-theory)))

;;
;;  INC
;;


(prove-lemma flags-interpretation-int-inc-overflow ()
  (let
    ((inc-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-inc)))))
    (implies
     (and (bvp a)
          (not (zerop (length a)))
          (v-set set-flags))
     (and (equal (store-resultp (cc-vc) inc-flags)
                 (integer-in-rangep (iplus (v-to-int a) 1)
                                    (length a)))
          (equal (store-resultp (cc-vs) inc-flags)
                 (not (integer-in-rangep (iplus (v-to-int a) 1)
                                         (length a)))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus
           v-alu-int-inc v-alu-int-adder v-alu-int-adder-overflowp
           v-alu-int-adder-output)
   (enable-theory set-flags-theory flags-theory)))

;;  Note that for integer increment, the Z flag is valid.
```

```
(prove-lemma flags-interpretation-int-inc-zero ()
  (let
    ((inc-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-inc)))))
    (implies
     (and (bvp a)
          (not (equal (length a) 0))
          (z-set set-flags))
     (and
      (equal (store-resultp (cc-ne) inc-flags)
             (not (equal (iplus (v-to-int a) 1) 0)))
      (equal (store-resultp (cc-eq) inc-flags)
             (equal (iplus (v-to-int a) 1) 0)))))

  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus izerop
           v-alu-int-inc v-alu-int-adder v-alu-int-adder-overflowp
           v-alu-int-adder-output)
   (enable-theory set-flags-theory flags-theory)))


(prove-lemma flags-interpretation-int-inc-negative ()
  (let
    ((inc-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-inc)))))
    (implies
     (and (bvp a)
          (not (equal (length a) 0))
          (not (equal (length a) 1))
          (v-set set-flags)
          (n-set set-flags))
     (and
      (equal (store-resultp (cc-ge) inc-flags)
             (not (negativep (iplus (v-to-int a) 1))))
      (equal (store-resultp (cc-lt) inc-flags)
             (negativep (iplus (v-to-int a) 1))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus
           v-alu-int-inc v-alu-int-adder v-alu-int-adder-overflowp
           v-alu-int-adder-output integer-in-rangep-iplus)
   (enable-theory set-flags-theory flags-theory)))
```

```
(prove-lemma flags-interpretation-int-inc-gt-0 ()
  (let
    ((inc-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-inc)))))
    (implies
     (and (bvp a)
          (not (equal (length a) 0))
          (not (equal (length a) 1))
          (n-set set-flags)
          (v-set set-flags)
          (z-set set-flags))
     (and
      (equal (store-resultp (cc-gt) inc-flags)
             (ilessp 0 (iplus 1 (v-to-int a))))
      (equal (store-resultp (cc-le) inc-flags)
             (ileq (iplus 1 (v-to-int a)) 0)))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus
           v-alu-int-inc v-alu-int-adder v-alu-int-adder-overflowp
           v-alu-int-adder-output
           integer-in-rangep-iplus izerop ileq)
   (enable-theory set-flags-theory flags-theory)))

;;
;;  ADDC
;;
```

```
(prove-lemma flags-interpretation-int-addc-overflow ()
  (let
    ((c (c-flag flags))
     (addc-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-addc)))))
    (implies
     (and (bv2p a b)
          (not (zerop (length a)))
          (v-set set-flags))
     (and (equal (store-resultp (cc-vc) addc-flags)
                 (integer-in-rangep (iplus (b-to-nat c)
                                           (iplus (v-to-int a) (v-to-int b)))
                                    (length a)))
          (equal (store-resultp (cc-vs) addc-flags)
                 (not (integer-in-rangep (iplus (b-to-nat c)
                                                (iplus (v-to-int a)
                                                       (v-to-int b)))
                                         (length a))))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus
           v-alu-int-inc v-alu-int-adder v-alu-int-adder-overflowp
           v-alu-int-adder-output)
   (enable-theory set-flags-theory flags-theory)))
```

```
(prove-lemma flags-interpretation-int-addc-negative ()
  (let
    ((c (c-flag flags))
     (addc-flags (update-flags flags set-flags
                               (v-alu (c-flag flags) a b (op-addc)))))
    (implies
     (and (bv2p a b)
          (not (equal (length a) 0))
          (not (equal (length a) 1))
          (v-set set-flags)
          (n-set set-flags))
     (and
      (equal (store-resultp (cc-lt) addc-flags)
             (negativep (iplus (b-to-nat c)
                               (iplus (v-to-int a) (v-to-int b)))))
      (equal (store-resultp (cc-ge) addc-flags)
             (not (negativep (iplus (b-to-nat c)
                                    (iplus (v-to-int a) (v-to-int b)))))))))

  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus
           v-alu-int-inc v-alu-int-adder v-alu-int-adder-overflowp
           v-alu-int-adder-output integer-in-rangep-iplus
           integer-in-rangep-iplus-carry)
   (enable-theory set-flags-theory flags-theory)))

; 256 Cases!
```

```
(prove-lemma flags-interpretation-int-addc-gt-0 ()
  (let
    ((c (c-flag flags))
     (addc-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-addc)))))
    (implies
     (and (bv2p a b)
          (not (equal (length a) 0))
          (not (equal (length a) 1))
          (n-set set-flags)
          (v-set set-flags)
          (z-set set-flags))
     (and
      (equal (store-resultp (cc-gt) addc-flags)
             (ilessp 0 (iplus (b-to-nat c) (iplus (v-to-int a) (v-to-int b)))))
      (equal (store-resultp (cc-le) addc-flags)
             (ileq (iplus (b-to-nat c)
                          (iplus (v-to-int a) (v-to-int b))) 0)))))
  ;;Hint
  ((use (integer-in-rangep-iplus$commuted
         (a (v-to-int a)) (b (v-to-int b)) (l (length a)))
        (integer-in-rangep-iplus-carry$commuted
         (a (v-to-int a)) (b (v-to-int b)) (l (length a))))
   (enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus
           v-alu-int-inc v-alu-int-adder v-alu-int-adder-overflowp
           v-alu-int-adder-output izerop ileq
           integer-in-rangep ineg)
   (enable-theory set-flags-theory flags-theory)))

;;
;;  ADD
;;
```

```
(prove-lemma flags-interpretation-int-add-overflow ()
  (let
    ((add-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-add)))))
    (implies
     (and (bv2p a b)
          (not (zerop (length a)))
          (v-set set-flags))
     (and (equal (store-resultp (cc-vc) add-flags)
                 (integer-in-rangep (iplus (v-to-int a) (v-to-int b))
                                    (length a)))
          (equal (store-resultp (cc-vs) add-flags)
                 (not (integer-in-rangep (iplus (v-to-int a) (v-to-int b))
                                         (length a)))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus
           v-alu-int-inc v-alu-int-adder v-alu-int-adder-overflowp
           v-alu-int-adder-output)
   (enable-theory set-flags-theory flags-theory)))


(prove-lemma flags-interpretation-int-add-negative ()
  (let
    ((add-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-add)))))
    (implies
     (and (bv2p a b)
          (not (equal (length a) 0))
          (v-set set-flags)
          (n-set set-flags))
     (and
      (equal (store-resultp (cc-lt) add-flags)
             (negativep (iplus (v-to-int a) (v-to-int b))))
      (equal (store-resultp (cc-ge) add-flags)
             (not (negativep (iplus (v-to-int a) (v-to-int b))))))))

  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus
           v-alu-int-inc v-alu-int-adder v-alu-int-adder-overflowp
           v-alu-int-adder-output integer-in-rangep-iplus)
   (enable-theory set-flags-theory flags-theory)))
```

```
(prove-lemma flags-interpretation-int-add-gt-0 ()
  (let
    ((add-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-add)))))
    (implies
     (and (bv2p a b)
          (not (equal (length a) 0))
          (not (equal (length a) 1))
          (n-set set-flags)
          (v-set set-flags)
          (z-set set-flags))
     (and
      (equal (store-resultp (cc-gt) add-flags)
             (ilessp 0 (iplus (v-to-int a) (v-to-int b))))
      (equal (store-resultp (cc-le) add-flags)
             (ileq (iplus (v-to-int a) (v-to-int b)) 0)))))
  ;;Hint
  ((use (integer-in-rangep-iplus$commuted
         (a (v-to-int a)) (b (v-to-int b)) (l (length a))))
   (enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus
           v-alu-int-inc v-alu-int-adder v-alu-int-adder-overflowp
           v-alu-int-adder-output izerop ileq
           integer-in-rangep ineg)
   (enable-theory set-flags-theory flags-theory)))

;;
;;  NEG
;;


(prove-lemma int-neg-range-implications (rewrite)
  (and
   (implies
    (and (negativep (v-to-int a))
         (not (integer-in-rangep (negative-guts (v-to-int a)) (length a))) )
    (v-negp (int-to-v (negative-guts (v-to-int a)) (length a))))
   (implies
    (and (not (negativep (v-to-int a)))
         (not (integer-in-rangep (minus (v-to-int a)) (length a))))
    (not (v-negp (int-to-v (minus (v-to-int a)) (length a)))))
   (implies
    (and
     (not (negativep (v-to-int a)))
     (not (integer-in-rangep (minus (v-to-int a)) (length a))) )
    (v-nzerop (int-to-v (minus (v-to-int a)) (length a)))))
  ;;Hint
  ((enable integer-in-rangep exp v-negp-as-bounds int-to-v remainder
           v-to-int)
   (enable-theory all-integer-defns)))
```

```
(prove-lemma flags-interpretation-int-neg-overflow ()
  (let
    ((neg-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-neg)))))
    (implies
     (and (bvp a)
          (not (zerop (length a)))
          (v-set set-flags))
     (and (equal (store-resultp (cc-vc) neg-flags)
                 (integer-in-rangep (ineg (v-to-int a)) (length a)))
          (equal (store-resultp (cc-vs) neg-flags)
                 (not (integer-in-rangep (ineg (v-to-int a)) (length a)))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus ineg idifference izerop
           v-alu-int-neg v-alu-int-subtracter v-alu-int-subtracter-overflowp)
   (enable-theory set-flags-theory flags-theory)))


(prove-lemma flags-interpretation-int-neg-negative ()
  (let
    ((neg-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-neg)))))
    (implies
     (and (bvp a)
          (not (equal (length a) 0))
          (not (equal (length a) 1))
          (v-set set-flags)
          (n-set set-flags))
     (and
      (equal (store-resultp (cc-lt) neg-flags)
             (negativep (ineg (v-to-int a))))
      (equal (store-resultp (cc-ge) neg-flags)
             (not (negativep (ineg (v-to-int a))))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus ineg idifference izerop integerp
           v-alu-int-neg v-alu-int-subtracter v-alu-int-subtracter-overflowp
           v-alu-int-subtracter-output integer-in-rangep-the-obvious-way)
   (enable-theory set-flags-theory flags-theory)))

;;  The Z flag is valid for negation.
```

```
(prove-lemma flags-interpretation-int-neg-zero ()
  (let
    ((neg-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-neg)))))
    (implies
     (and (bvp a)
          (not (equal (length a) 0))
          (z-set set-flags))
     (and
      (equal (store-resultp (cc-ne) neg-flags)
             (not (equal (ineg (v-to-int a)) 0)))
      (equal (store-resultp (cc-eq) neg-flags)
             (equal (ineg (v-to-int a)) 0)))))

  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus ineg idifference izerop integerp
           v-alu-int-neg v-alu-int-subtracter v-alu-int-subtracter-overflowp
           v-alu-int-subtracter-output integer-in-rangep-the-obvious-way)
   (enable-theory set-flags-theory flags-theory)))



(prove-lemma flags-interpretation-int-neg-gt-0 ()
  (let
    ((neg-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-neg)))))
    (implies
     (and (bvp a)
          (not (equal (length a) 0))
          (not (equal (length a) 1))
          (n-set set-flags)
          (v-set set-flags)
          (z-set set-flags))
     (and
      (equal (store-resultp (cc-gt) neg-flags)
             (ilessp 0 (ineg (v-to-int a))))
      (equal (store-resultp (cc-le) neg-flags)
             (ileq (ineg (v-to-int a)) 0)))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus ineg idifference izerop integerp ileq
           v-alu-int-neg v-alu-int-subtracter v-alu-int-subtracter-overflowp
           v-alu-int-subtracter-output integer-in-rangep-the-obvious-way)
   (enable-theory set-flags-theory flags-theory)))

;;
;;   DEC
;;
```

```
(prove-lemma int-dec-rangep-lemmas (rewrite)
  (and
   (implies
    (not (integer-in-rangep (sub1 (v-to-int a)) (length a)))
    (v-nzerop (int-to-v (sub1 (v-to-int a)) (length a))))
   (implies
    (and (not (equal (length a) 1))
         (not (integer-in-rangep (minus (add1 (negative-guts (v-to-int a))))
                                 (length a))))
    (v-nzerop (int-to-v (minus (add1 (negative-guts (v-to-int a))))
                        (length a))))
   (implies
    (and
     (not (negativep (iplus -1 (v-to-int a))))
     (v-nzerop (int-to-v (iplus -1 (v-to-int a)) (length a))))
    (not (equal (iplus -1 (v-to-int a)) 0)))
   (implies
    (and
     (not (zerop (length a)))
     (not (negativep (iplus -1 (v-to-int a))))
     (not (v-nzerop (int-to-v (iplus -1 (v-to-int a)) (length a)))))
    (equal (iplus -1 (v-to-int a)) 0)))
  ;;Hint
  ((enable integer-in-rangep exp v-negp-as-bounds int-to-v remainder
           v-to-int)
   (enable-theory all-integer-defns)))


(prove-lemma flags-interpretation-int-dec-overflow ()
  (let
    ((dec-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-dec)))))
    (implies
     (and (bvp a)
          (not (zerop (length a)))
          (v-set set-flags))
     (and (equal (store-resultp (cc-vc) dec-flags)
                 (integer-in-rangep (idifference (v-to-int a) 1) (length a)))
          (equal (store-resultp (cc-vs) dec-flags)
                 (not (integer-in-rangep (idifference (v-to-int a) 1)
                                         (length a)))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus ineg idifference izerop integerp
           v-alu-int-dec v-alu-int-subtracter
           v-alu-int-subtracter-overflowp)
   (enable-theory set-flags-theory flags-theory)))
```

```
(prove-lemma flags-interpretation-int-dec-negative ()
  (let
    ((dec-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-dec)))))
    (implies
     (and (bvp a)
          (not (equal (length a) 0))
          (v-set set-flags)
          (n-set set-flags))
     (and
      (equal (store-resultp (cc-lt) dec-flags)
             (negativep (idifference (v-to-int a) 1)))
      (equal (store-resultp (cc-ge) dec-flags)
             (not (negativep (idifference (v-to-int a) 1)))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus ineg idifference izerop integerp
           v-alu-int-dec v-alu-int-subtracter v-alu-int-subtracter-overflowp
           v-alu-int-subtracter-output integer-in-rangep-iplus)
   (enable-theory set-flags-theory flags-theory)))

;;  Z flag is valid for decrement.


(prove-lemma flags-interpretation-int-dec-zero ()
  (let
    ((dec-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-dec)))))
    (implies
     (and (bvp a)
          (not (equal (length a) 0))
          (not (equal (length a) 1))
          (z-set set-flags))
     (and
      (equal (store-resultp (cc-ne) dec-flags)
             (not (equal (idifference (v-to-int a) 1) 0)))
      (equal (store-resultp (cc-eq) dec-flags)
             (equal (idifference (v-to-int a) 1) 0)))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus ineg idifference izerop integerp iplus
           v-alu-int-dec v-alu-int-subtracter v-alu-int-subtracter-overflowp
           v-alu-int-subtracter-output)
   (enable-theory set-flags-theory flags-theory)))
```

```
(prove-lemma flags-interpretation-int-dec-gt-0 ()
  (let
    ((dec-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-dec)))))
    (implies
     (and (bvp a)
          (not (equal (length a) 0))
          (not (equal (length a) 1))
          (n-set set-flags)
          (v-set set-flags)
          (z-set set-flags))
     (and
      (equal (store-resultp (cc-gt) dec-flags)
             (ilessp 0 (idifference (v-to-int a) 1)))
      (equal (store-resultp (cc-le) dec-flags)
             (ileq (idifference (v-to-int a) 1) 0))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb fix-int ilessp integerp-iplus
           ineg idifference izerop integerp ileq
           v-alu-int-dec v-alu-int-subtracter v-alu-int-subtracter-overflowp
           v-alu-int-subtracter-output integer-in-rangep-iplus)
   (enable-theory set-flags-theory flags-theory)))

;;
;;  SUBB
;;


(prove-lemma flags-interpretation-int-subb-overflow ()
  (let
    ((c (c-flag flags))
     (subb-flags (update-flags flags set-flags
                               (v-alu (c-flag flags) a b (op-subb)))))
    (implies
     (and (bv2p a b)
          (not (zerop (length a)))
          (v-set set-flags))
     (and (equal (store-resultp (cc-vc) subb-flags)
                 (integer-in-rangep
                  (idifference (v-to-int b)
                               (iplus (v-to-int a) (b-to-nat c)))
                  (length a)))
          (equal (store-resultp (cc-vs) subb-flags)
                 (not (integer-in-rangep
                       (idifference (v-to-int b)
                                    (iplus (v-to-int a) (b-to-nat c)))
                       (length a)))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb fix-int ilessp integerp-iplus
           v-alu-int-subtracter v-alu-int-subtracter-overflowp)
   (enable-theory set-flags-theory flags-theory)))
```

```
(prove-lemma flags-interpretation-int-subb-negative ()
  (let
    ((c (c-flag flags))
     (subb-flags (update-flags flags set-flags
                               (v-alu (c-flag flags) a b (op-subb)))))
    (implies
     (and (bv2p a b)
          (not (equal (length a) 0))
          (not (equal (length a) 1))
          (v-set set-flags)
          (n-set set-flags))
     (and
      (equal (store-resultp (cc-lt) subb-flags)
             (negativep (idifference (v-to-int b)
                                     (iplus (v-to-int a) (b-to-nat c)))))
      (equal (store-resultp (cc-ge) subb-flags)
             (not (negativep
                   (idifference (v-to-int b)
                                (iplus (v-to-int a) (b-to-nat c))))))))))

  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb fix-int ilessp integerp-iplus idifference
           v-alu-int-subtracter v-alu-int-subtracter-overflowp
           v-alu-int-subtracter-output integer-in-rangep-iplus-ineg
           integer-in-rangep-iplus-ineg-carry)
   (enable-theory set-flags-theory flags-theory)))
```

```
(prove-lemma flags-interpretation-int-subb-gt-0 ()
  (let
    ((c (c-flag flags))
     (subb-flags (update-flags flags set-flags
                               (v-alu (c-flag flags) a b (op-subb)))))
    (implies
     (and (bv2p a b)
          (not (equal (length a) 0))
          (not (equal (length a) 1))
          (n-set set-flags)
          (v-set set-flags)
          (z-set set-flags))
     (and
      (equal (store-resultp (cc-gt) subb-flags)
             (ilessp 0 (idifference (v-to-int b)
                                    (iplus (v-to-int a) (b-to-nat c)))))
      (equal (store-resultp (cc-le) subb-flags)
             (ileq (idifference (v-to-int b)
                                (iplus (v-to-int a) (b-to-nat c))) 0)))))
  ;;Hint
  ((use (integer-in-rangep-iplus-ineg-carry$commuted
         (a (v-to-int a)) (b (v-to-int b)) (c (b-to-nat (c-flag flags)))
         (l (length a)))
        (integer-in-rangep-iplus-ineg$commuted
         (a (v-to-int a)) (b (v-to-int b)) (l (length a))))
   (enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb fix-int ilessp integerp-iplus
           v-alu-int-subtracter v-alu-int-subtracter-overflowp
           v-alu-int-subtracter-output
           izerop ileq idifference integer-in-rangep ineg)
   (enable-theory set-flags-theory flags-theory)))

;;
;;    SUB
;;
```

```
(prove-lemma flags-interpretation-int-sub-overflow ()
  (let
    ((sub-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-sub)))))
    (implies
     (and (bv2p a b)
          (not (zerop (length a)))
          (v-set set-flags))
     (and (equal (store-resultp (cc-vc) sub-flags)
                 (integer-in-rangep (idifference (v-to-int b) (v-to-int a))
                                    (length a)))
          (equal (store-resultp (cc-vs) sub-flags)
                 (not (integer-in-rangep (idifference (v-to-int b)
                                                      (v-to-int a))
                                         (length a)))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb fix-int ilessp integerp-iplus
           v-alu-int-subtracter v-alu-int-subtracter-overflowp)
   (enable-theory set-flags-theory flags-theory)))


(prove-lemma flags-interpretation-int-sub-negative ()
  (let
    ((sub-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-sub)))))
    (implies
     (and (bv2p a b)
          (not (equal (length a) 0))
          (not (equal (length a) 1))
          (v-set set-flags)
          (n-set set-flags))
     (and
      (equal (store-resultp (cc-lt) sub-flags)
             (negativep (idifference (v-to-int b) (v-to-int a))))
      (equal (store-resultp (cc-ge) sub-flags)
             (not (negativep (idifference (v-to-int b) (v-to-int a))))))))

  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb fix-int ilessp integerp-iplus idifference
           v-alu-int-subtracter v-alu-int-subtracter-overflowp
           v-alu-int-subtracter-output integer-in-rangep-iplus-ineg)
   (enable-theory set-flags-theory flags-theory)))
```

```
(prove-lemma flags-interpretation-int-sub-gt-0 ()
  (let
    ((sub-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-sub)))))
    (implies
     (and (bv2p a b)
          (not (equal (length a) 0))
          (not (equal (length a) 1))
          (n-set set-flags)
          (v-set set-flags)
          (z-set set-flags))
     (and
      (equal (store-resultp (cc-gt)sub-flags)
             (ilessp 0 (idifference (v-to-int b) (v-to-int a))))
      (equal (store-resultp (cc-le) sub-flags)
             (ileq (idifference (v-to-int b) (v-to-int a)) 0)))))
  ;;Hint
  ((use (integer-in-rangep-iplus-ineg$commuted
         (a (v-to-int a)) (b (v-to-int b)) (l (length a))))
   (enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb fix-int ilessp integerp-iplus
           v-alu-int-subtracter v-alu-int-subtracter-overflowp
           v-alu-int-subtracter-output
           izerop ileq idifference integer-in-rangep ineg)
   (enable-theory set-flags-theory flags-theory)))

;;
;;   ASR
;;


(prove-lemma equal-idiv-2-0 (rewrite)
  (equal (equal (idiv i 2) 0)
         (or (izerop i) (equal i 1)))
  ;;Hint
  ((enable idiv fix-int izerop integerp)))


(prove-lemma integer-in-rangep-idiv (rewrite)
  (implies
   (integer-in-rangep i l)
   (integer-in-rangep (idiv i 2) l))
  ;;Hint
  ((enable integer-in-rangep exp v-negp-as-bounds int-to-v remainder
           v-to-int quotient exp
            quotient-lessp quotient-stuff not-lessp-quotient)
   (enable-theory all-integer-defns)))
```

```
(prove-lemma flags-interpretation-int-asr-zero ()
  (let
    ((asr-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-asr)))))
    (implies
     (and (bvp a)
          (not (zerop (length a)))
          (z-set set-flags))
     (and (equal (store-resultp (cc-ne) asr-flags)
                 (not (equal (idiv (v-to-int a) 2) 0)))
          (equal (store-resultp (cc-eq) asr-flags)
                 (equal (idiv (v-to-int a) 2) 0)))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu integerp-idiv
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus izerop
           v-alu-int-asr v-alu-int-asr-output)
   (enable-theory set-flags-theory flags-theory)))


(prove-lemma equal-sub1-plus-x-x (rewrite)
  (equal (equal (sub1 (plus x x)) 0)
         (zerop x)))


(prove-lemma remainder-difference-plus-x-x-plus-y-y (rewrite)
   (equal (remainder (difference (plus x x) (plus y y)) 2) 0)
  ;;Hint
  ((enable remainder difference plus)
   (induct (difference x y))))


(prove-lemma remainder-difference-plus-x-x-add1-plus-y-y (rewrite)
  (implies
   (lessp (add1 (plus y y)) (plus x x))
   (equal (remainder (difference (plus x x) (add1 (plus y y))) 2) 1))
  ;;Hint
  ((enable remainder difference plus)
   (induct (difference x y))))


(prove-lemma quotient-difference-plus-x-x-plus-y-y-2 (rewrite)
  (equal
   (quotient (difference (plus x x) (plus y y)) 2)
   (difference x y))
  ;;Hint
  ((enable quotient difference plus)
   (induct (difference x y))))
```

```
(prove-lemma quotient-difference-plus-x-x-add1-plus-y-y-2 (rewrite)
  (equal
   (quotient (difference (plus x x) (add1 (plus y y))) 2)
   (sub1 (difference x y)))
  ;;Hint
  ((enable quotient difference plus)
   (induct (difference x y))))


(prove-lemma quotient-add1-plus-x-x-2 (rewrite)
  (equal (quotient (add1 (plus x x)) 2)
         (fix x))
  ;;Hint
  ((enable quotient plus)))


(prove-lemma imod-v-to-int-2 (rewrite)
  (implies
   (listp a)
   (equal (imod (v-to-int a) 2)
          (b-to-nat (car a))))
  ;;Hint
  ((enable imod v-to-int v-to-nat b-to-nat exp quotient-plus-x-x-2)
   (enable-theory all-integer-defns)
   (expand (v-to-nat a))))


(prove-lemma flags-interpretation-int-asr-carry ()
  (let
    ((asr-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-asr)))))
    (implies
     (and (bvp a)
          (not (zerop (length a)))
          (c-set set-flags))
     (and (equal (store-resultp (cc-cc) asr-flags)
                 (equal (imod (v-to-int a) 2) 0))
          (equal (store-resultp (cc-cs) asr-flags)
                 (not (equal (imod (v-to-int a) 2) 0))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus izerop
           v-alu-int-asr v-alu-int-asr-output)
   (enable-theory set-flags-theory flags-theory)))

;  Note that for ASR the V bit is always F.
```

```
(prove-lemma flags-interpretation-int-asr-negative ()
  (let
    ((asr-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-asr)))))
    (implies
     (and (bvp a)
          (not (zerop (length a))))
     (and
      (implies
       (n-set set-flags)
       (and
        (equal (store-resultp (cc-pl) asr-flags)
               (not (negativep (idiv (v-to-int a) 2))))
        (equal (store-resultp (cc-mi) asr-flags)
               (negativep (idiv (v-to-int a) 2)))))
      (implies
       (and (n-set set-flags)
            (v-set set-flags))
       (and
        (equal (store-resultp (cc-ge) asr-flags)
               (not (negativep (idiv (v-to-int a) 2))))
        (equal (store-resultp (cc-lt) asr-flags)
               (negativep (idiv (v-to-int a) 2))))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus
           v-alu-int-asr v-alu-int-asr-output
           integerp-idiv integer-in-rangep-the-obvious-way)
   (enable-theory set-flags-theory flags-theory)))
```

```
(prove-lemma flags-interpretation-int-asr-gt-0 ()
  (let
    ((asr-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-asr)))))
    (implies
     (and (bvp a)
          (not (zerop (length a)))
          (v-set set-flags)
          (n-set set-flags)
          (z-set set-flags))
     (and
      (equal (store-resultp (cc-gt) asr-flags)
             (ilessp 0 (idiv (v-to-int a) 2)))
      (equal (store-resultp (cc-le) asr-flags)
             (ileq (idiv (v-to-int a) 2) 0)))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus
           v-alu-int-asr v-alu-int-asr-output
           integerp-idiv ilessp ileq izerop
           integer-in-rangep-the-obvious-way)
   (enable-theory set-flags-theory flags-theory)))


(prove-lemma flags-interpretation-int-asr-overflow ()
  ;;  We prove that our decision to set V = F for ASR is well founded.
  (let
    ((asr-flags (update-flags flags set-flags
                              (v-alu (c-flag flags) a b (op-asr)))))
    (implies
     (and (bvp a)
          (v-set set-flags))
     (and
      (equal (store-resultp (cc-vc) asr-flags)
             (integer-in-rangep (idiv (v-to-int a) 2) (length a)))
      (equal (store-resultp (cc-vs) asr-flags)
             (not (integer-in-rangep (idiv (v-to-int a) 2) (length a)))))))
  ;;Hint
  ((enable set-flags update-flags store-resultp v-alu
           c v n cv bv b-to-nat zb
           fix-int ilessp integerp-iplus
           v-alu-int-asr v-alu-int-asr-output
           integerp-idiv ilessp ileq izerop
           integer-in-rangep-the-obvious-way)
   (enable-theory set-flags-theory flags-theory)))
```

14.73 "more-alu-interpretation.events"

```
;;;  Copyright (C) 1990-1994 Computational Logic, Inc.  All Rights
;;;  Reserved.  See the file LICENSE in this directory for the
;;;  complete license agreement.

;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;;;
;;;    MORE-ALU-INTERPRETATION.EVENTS
;;;
;;;    The lemmas in this file simply restate what was proved in
;;;    "alu-interpretation.events" in an easily readable way.
;;;
;;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defn v-neg (a)
  (v-subtracter-output f a (nat-to-v 0 (length a))))

;;;  Lemmas that explicate the BV, C, V, and Z of (V-ALU c a b).


(prove-lemma bv-cvzbv (rewrite)
  (equal (bv (cvzbv c v bv))
         bv)
  ;;Hint
  ((enable bv cvzbv)))


(prove-lemma c-cvzbv (rewrite)
  (equal (c (cvzbv c v bv))
         c)
  ;;Hint
  ((enable c cvzbv)))


(prove-lemma v-cvzbv (rewrite)
  (equal (v (cvzbv c v bv))
         v)
  ;;Hint
  ((enable v cvzbv)))


(prove-lemma zb-cvzbv (rewrite)
  (equal (zb (cvzbv c v bv))
         (v-zerop bv))
  ;;Hint
  ((enable zb cvzbv)))

;;;  An easily understandable form of the V-ALU specification.
```

```
(prove-lemma bv-v-alu ()
  (equal
   (bv (v-alu c a b op))
   (cond
    ((equal op #v0000) (v-buf a))                    ;MOVE
    ((equal op #v0001) (v-inc a))                    ;INC
    ((equal op #v0010) (v-adder-output c a b))       ;ADDC
    ((equal op #v0011) (v-adder-output f a b))       ;ADD
    ((equal op #v0100) (v-neg a))                    ;NEG
    ((equal op #v0101) (v-dec a))                    ;DEC
    ((equal op #v0110) (v-subtracter-output c a b)) ;SUBB
    ((equal op #v0111) (v-subtracter-output f a b)) ;SUB
    ((equal op #v1000) (v-ror a c))                  ;ROR
    ((equal op #v1001) (v-asr a))                    ;ASR
    ((equal op #v1010) (v-lsr a))                    ;LSR
    ((equal op #v1011) (v-xor a b))                  ;XOR
    ((equal op #v1100) (v-or  a b))                  ;OR
    ((equal op #v1101) (v-and a b))                  ;AND
    ((equal op #v1110) (v-not a))                    ;NOT
    (t                 (v-buf a))))                  ;M15
  ;;Hint
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable v-alu bv-cvzbv
           cvzbv-inc cvzbv-v-adder cvzbv-neg cvzbv-dec cvzbv-v-subtracter
           cvzbv-v-ror cvzbv-v-asr cvzbv-v-lsr cvzbv-v-not
           v-inc v-dec v-neg)))


(prove-lemma c-v-alu ()
  (and
   (implies
    (or (equal op (op-move)) (equal op (op-xor)) (equal op (op-or))
        (equal op (op-and))  (equal op (op-not)) (equal op (op-m15)))
    (equal (c (v-alu c a b op))
           f))
   (implies
    (and
     (listp a)
     (or (equal op (op-ror)) (equal op (op-asr)) (equal op (op-lsr))))
    (equal (c (v-alu c a b op))
           (nth 0 a))))
  ;;Hint
  ((disable-theory t)
   (enable-theory ground-zero opcode-theory)
   (enable v-alu c-cvzbv
           cvzbv-inc cvzbv-v-adder cvzbv-neg cvzbv-dec cvzbv-v-subtracter
           cvzbv-v-ror cvzbv-v-asr cvzbv-v-lsr cvzbv-v-not
           v-inc v-dec v-neg)))
```

```
(prove-lemma v-v-alu ()
  (implies
   (or
    (equal op (op-move)) (equal op (op-ror)) (equal op (op-asr))
    (equal op (op-asr))  (equal op (op-lsr)) (equal op (op-xor))
    (equal op (op-or))   (equal op (op-and)) (equal op (op-not))
    (equal op (op-m15)))
   (equal (v (v-alu c a b op))
          f))
  ;;Hint
  ((disable-theory t)
   (enable-theory ground-zero opcode-theory)
   (enable v-alu v-cvzbv
           cvzbv-inc cvzbv-v-adder cvzbv-neg cvzbv-dec cvzbv-v-subtracter
           cvzbv-v-ror cvzbv-v-asr cvzbv-v-lsr cvzbv-v-not
           v-inc v-dec v-neg)))

;;;  Matt introduced these lemmas without the REWRITE option.


(prove-lemma v-alu-correct-nat-rewrite (rewrite)
  (implies (bv2p a b)
           (equal (v-alu c a b op)
                  (v-alu-nat c a b op)))
  ;;Hint
  ((disable v-alu v-alu-nat)
   (use (v-alu-correct-nat))))


(disable v-alu-correct-nat-rewrite)


(prove-lemma v-alu-correct-int-rewrite (rewrite)
  (implies
   (and (bv2p a b)
        (not (equal (length a) 0)))
   (equal (v-alu c a b op)
          (v-alu-int c a b op)))
  ;;Hint
  ((disable v-alu v-alu-int)
   (use (v-alu-correct-int))))


(disable v-alu-correct-int-rewrite)


(prove-lemma lessp-quotient-test (rewrite)
  (implies
   (lessp n m)
   (equal (lessp (quotient n k) m)
          t))
  ;;Hint
  ((enable quotient lessp)))
```

```
(disable lessp-quotient-test)


(prove-lemma nat-to-v-as-remainder (rewrite)
  (equal (nat-to-v (remainder n (exp 2 l)) l)
         (nat-to-v n l))
  ;;Hint
  ((enable nat-to-v quotient-remainder remainder-exp quotient-exp
           remainder-remainder)
   (expand (nat-to-v (remainder n (exp 2 l)) l))
   (induct (nat-to-v n l))))


(prove-lemma nat-to-v-of-v-to-nat* (rewrite)
  (implies
   (and (bvp v)
        (equal l (length v)))
   (equal (nat-to-v (v-to-nat v) l)
          v)))


(disable nat-to-v-of-v-to-nat*)

;;;  Here is an easily readable form of the natural ALU.
```

```
(prove-lemma bv-v-alu-as-natural ()
  (let
    ((cn (b-to-nat c))
     (an (v-to-nat a))
     (bn (v-to-nat b)))
    (implies
     (and (bv2p a b)
          (equal (length a) 32))
     (equal
      (bv (v-alu c a b op))
      (cond
       ((equal op (op-inc))  (nat-to-v (plus 1 an) 32))
       ((equal op (op-addc)) (nat-to-v (plus cn (plus an bn)) 32))
       ((equal op (op-add))  (nat-to-v (plus an bn) 32))
       ((equal op (op-dec))  (nat-to-v (difference (plus an (exp 2 32))
                                                   1)
                                       32))
       ((equal op (op-subb)) (nat-to-v (difference (plus bn (exp 2 32))
                                                   (plus an cn))
                                       32))
       ((equal op (op-sub))  (nat-to-v (difference (plus bn (exp 2 32))
                                                   an)
                                       32))
       ((equal op (op-lsr))  (nat-to-v (quotient an 2) 32))
       (t                    (bv (v-alu c a b op)))))))
  ;;Hint
  ((disable cvzbv bv)
   (enable bv-cvzbv nat-to-v-of-v-to-nat*
           remainder-remainder remainder-noop lessp-quotient-test
           v-to-nat-of-nat-to-v *1*b-to-nat
           v-alu-correct-nat-rewrite v-alu-nat
           v-alu-nat-int-buf v-alu-nat-inc v-alu-nat-dec
           v-alu-nat-adder v-alu-nat-adder-output
           v-alu-nat-subtracter v-alu-nat-subtracter-output
           v-alu-nat-lsr v-alu-nat-lsr-output)))


(prove-lemma fix-int-int-to-v (rewrite)
  (equal (fix-int (v-to-int v))
         (v-to-int v))
  ;;Hint
  ((enable fix-int v-to-int integerp idifference iplus)))


(prove-lemma int-to-v-of-v-to-int (rewrite)
  (implies
   (and (equal l (length v))
        (bvp v))
   (equal (int-to-v (v-to-int v) l)
          v))
  ;;Hint
  ((enable int-to-v v-to-int)
   (enable-theory all-integer-defns)))
```

```
(prove-lemma ineg-as-idifference (rewrite)
  (equal (idifference 0 i)
         (ineg i))
  ;;Hint
  ((enable-theory all-integer-defns)))


;;;   Here is an easily readable form of the integer ALU.


(prove-lemma bv-v-alu-as-integer ()
  (let
    ((cn (b-to-nat c))
     (an (v-to-int a))
     (bn (v-to-int b)))
    (implies
     (and (bv2p a b)
          (equal (length a) 32))
     (equal
      (bv (v-alu c a b op))
      (cond
       ((equal op (op-inc))  (int-to-v (iplus 1 an) 32))
       ((equal op (op-addc)) (int-to-v (iplus cn (iplus an bn)) 32))
       ((equal op (op-add))  (int-to-v (iplus an bn) 32))
       ((equal op (op-neg))  (int-to-v (ineg an) 32))
       ((equal op (op-dec))  (int-to-v (idifference an 1) 32))
       ((equal op (op-subb)) (int-to-v (idifference bn (iplus an cn)) 32))
       ((equal op (op-sub))  (int-to-v (idifference bn an) 32))
       ((equal op (op-asr))  (int-to-v (idiv an 2) 32))
       (t                    (bv (v-alu c a b op)))))))
  ;;Hint
  ((disable cvzbv bv)
   (enable bv-cvzbv
           remainder-remainder remainder-noop lessp-quotient-test
           *1*b-to-nat *1*v-to-int *1*fix-int
           fix-int-iplus fix-int-ineg fix-int-idifference
           fix-int-int-to-v
           v-alu-correct-int-rewrite v-alu-int v-alu-int-neg
           v-alu-nat-int-buf v-alu-int-inc v-alu-int-dec
           v-alu-int-adder v-alu-int-adder-output
           v-alu-int-subtracter v-alu-int-subtracter-output
           v-alu-int-asr v-alu-int-asr-output)))
```

Chapter 15

THE FM9001 NETLIST

Here is the netlist that we supplied to LSI Logic to fabricate the FM9001. Although this netlist includes a specification of all the internal wiring it does not specify three matters necessary for fabrication, namely chip size, package used, and I/O pad to package pinout assignment. Additional material not presented here that we were obliged to supply to LSI Logic includes simulation control files, functional and post manufacture test vectors, and a rough two-dimensional layout of the top-level modules.

```
COMPILE;
DIRECTORY MASTER;

MODULE CHIP;
INPUTS CLK,TI,TE,DTACK-,RESET-,HOLD-,DISABLE-REGFILE-,TEST-REGFILE-,
       PC-REG-IN.0,PC-REG-IN.1,PC-REG-IN.2,PC-REG-IN.3,DATA-BUS.0,DATA-BUS.1,
       DATA-BUS.2,DATA-BUS.3,DATA-BUS.4,DATA-BUS.5,DATA-BUS.6,DATA-BUS.7,
       DATA-BUS.8,DATA-BUS.9,DATA-BUS.10,DATA-BUS.11,DATA-BUS.12,DATA-BUS.13,
       DATA-BUS.14,DATA-BUS.15,DATA-BUS.16,DATA-BUS.17,DATA-BUS.18,
       DATA-BUS.19,DATA-BUS.20,DATA-BUS.21,DATA-BUS.22,DATA-BUS.23,
       DATA-BUS.24,DATA-BUS.25,DATA-BUS.26,DATA-BUS.27,DATA-BUS.28,
       DATA-BUS.29,DATA-BUS.30,DATA-BUS.31;
OUTPUTS PO,TO,TIMING,HDACK-,RW-,STROBE-,ADDR-OUT.0,ADDR-OUT.1,ADDR-OUT.2,
        ADDR-OUT.3,ADDR-OUT.4,ADDR-OUT.5,ADDR-OUT.6,ADDR-OUT.7,ADDR-OUT.8,
        ADDR-OUT.9,ADDR-OUT.10,ADDR-OUT.11,ADDR-OUT.12,ADDR-OUT.13,
        ADDR-OUT.14,ADDR-OUT.15,ADDR-OUT.16,ADDR-OUT.17,ADDR-OUT.18,
        ADDR-OUT.19,ADDR-OUT.20,ADDR-OUT.21,ADDR-OUT.22,ADDR-OUT.23,
        ADDR-OUT.24,ADDR-OUT.25,ADDR-OUT.26,ADDR-OUT.27,ADDR-OUT.28,
        ADDR-OUT.29,ADDR-OUT.30,ADDR-OUT.31,DATA-BUS.0,DATA-BUS.1,DATA-BUS.2,
        DATA-BUS.3,DATA-BUS.4,DATA-BUS.5,DATA-BUS.6,DATA-BUS.7,DATA-BUS.8,
        DATA-BUS.9,DATA-BUS.10,DATA-BUS.11,DATA-BUS.12,DATA-BUS.13,
        DATA-BUS.14,DATA-BUS.15,DATA-BUS.16,DATA-BUS.17,DATA-BUS.18,
        DATA-BUS.19,DATA-BUS.20,DATA-BUS.21,DATA-BUS.22,DATA-BUS.23,
        DATA-BUS.24,DATA-BUS.25,DATA-BUS.26,DATA-BUS.27,DATA-BUS.28,
        DATA-BUS.29,DATA-BUS.30,DATA-BUS.31,FLAGS.0,FLAGS.1,FLAGS.2,FLAGS.3,
        CNTL-STATE.0,CNTL-STATE.1,CNTL-STATE.2,CNTL-STATE.3,CNTL-STATE.4,
        I-REG.28,I-REG.29,I-REG.30,I-REG.31;
```

```
LEVEL FUNCTION;
DEFINE
BODY(I-TO,I-TIMING,I-HDACK-,I-EN-ADDR-OUT-,I-RW-,I-STROBE-,I-ADDR-OUT.0,
     I-ADDR-OUT.1,I-ADDR-OUT.2,I-ADDR-OUT.3,I-ADDR-OUT.4,I-ADDR-OUT.5,
     I-ADDR-OUT.6,I-ADDR-OUT.7,I-ADDR-OUT.8,I-ADDR-OUT.9,I-ADDR-OUT.10,
     I-ADDR-OUT.11,I-ADDR-OUT.12,I-ADDR-OUT.13,I-ADDR-OUT.14,I-ADDR-OUT.15,
     I-ADDR-OUT.16,I-ADDR-OUT.17,I-ADDR-OUT.18,I-ADDR-OUT.19,I-ADDR-OUT.20,
     I-ADDR-OUT.21,I-ADDR-OUT.22,I-ADDR-OUT.23,I-ADDR-OUT.24,I-ADDR-OUT.25,
     I-ADDR-OUT.26,I-ADDR-OUT.27,I-ADDR-OUT.28,I-ADDR-OUT.29,I-ADDR-OUT.30,
     I-ADDR-OUT.31,I-DATA-OUT.0,I-DATA-OUT.1,I-DATA-OUT.2,I-DATA-OUT.3,
     I-DATA-OUT.4,I-DATA-OUT.5,I-DATA-OUT.6,I-DATA-OUT.7,I-DATA-OUT.8,
     I-DATA-OUT.9,I-DATA-OUT.10,I-DATA-OUT.11,I-DATA-OUT.12,I-DATA-OUT.13,
     I-DATA-OUT.14,I-DATA-OUT.15,I-DATA-OUT.16,I-DATA-OUT.17,I-DATA-OUT.18,
     I-DATA-OUT.19,I-DATA-OUT.20,I-DATA-OUT.21,I-DATA-OUT.22,I-DATA-OUT.23,
     I-DATA-OUT.24,I-DATA-OUT.25,I-DATA-OUT.26,I-DATA-OUT.27,I-DATA-OUT.28,
     I-DATA-OUT.29,I-DATA-OUT.30,I-DATA-OUT.31,I-FLAGS.0,I-FLAGS.1,I-FLAGS.2,
     I-FLAGS.3,I-CNTL-STATE.0,I-CNTL-STATE.1,I-CNTL-STATE.2,I-CNTL-STATE.3,
     I-CNTL-STATE.4,I-I-REG.28,I-I-REG.29,I-I-REG.30,I-I-REG.31)
  = CHIP-MODULE(I-CLK,I-TI,I-TE,I-DTACK-,I-RESET-,I-HOLD-,
                I-DISABLE-REGFILE-,I-TEST-REGFILE-,I-PC-REG.0,I-PC-REG.1,
                I-PC-REG.2,I-PC-REG.3,I-DATA-IN.0,I-DATA-IN.1,I-DATA-IN.2,
                I-DATA-IN.3,I-DATA-IN.4,I-DATA-IN.5,I-DATA-IN.6,I-DATA-IN.7,
                I-DATA-IN.8,I-DATA-IN.9,I-DATA-IN.10,I-DATA-IN.11,
                I-DATA-IN.12,I-DATA-IN.13,I-DATA-IN.14,I-DATA-IN.15,
                I-DATA-IN.16,I-DATA-IN.17,I-DATA-IN.18,I-DATA-IN.19,
                I-DATA-IN.20,I-DATA-IN.21,I-DATA-IN.22,I-DATA-IN.23,
                I-DATA-IN.24,I-DATA-IN.25,I-DATA-IN.26,I-DATA-IN.27,
                I-DATA-IN.28,I-DATA-IN.29,I-DATA-IN.30,I-DATA-IN.31);
PLUS-5(B-TRUE-P) = VDD();
CLOCK-PAD(I-CLK,CLK-PO) = &DRVT8(CLK,B-TRUE-P);
TI-PAD(I-TI,TI-PO) = &TLCHT(TI,B-TRUE-P);
TE-PAD(I-TE,TE-PO) = &TLCHT(TE,TI-PO);
DTACK-PAD(I-DTACK-,DTACK-PO) = &TLCHT(DTACK-,TE-PO);
RESET-PAD(I-RESET-,RESET-PO) = &TLCHT(RESET-,DTACK-PO);
HOLD-PAD(I-HOLD-,HOLD-PO) = &TLCHT(HOLD-,RESET-PO);
DISABLE-REGFILE-PAD(I-DISABLE-REGFILE-,DISABLE-REGFILE-PO)
  = &TLCHT(DISABLE-REGFILE-,HOLD-PO);
TEST-REGFILE-PAD(I-TEST-REGFILE-,TEST-REGFILE-PO)
  = &TLCHT(TEST-REGFILE-,DISABLE-REGFILE-PO);
DATA-BUS-PADS(DATA-BUS-PO,DATA-BUS.0,DATA-BUS.1,DATA-BUS.2,DATA-BUS.3,
              DATA-BUS.4,DATA-BUS.5,DATA-BUS.6,DATA-BUS.7,DATA-BUS.8,
              DATA-BUS.9,DATA-BUS.10,DATA-BUS.11,DATA-BUS.12,DATA-BUS.13,
              DATA-BUS.14,DATA-BUS.15,DATA-BUS.16,DATA-BUS.17,DATA-BUS.18,
              DATA-BUS.19,DATA-BUS.20,DATA-BUS.21,DATA-BUS.22,DATA-BUS.23,
              DATA-BUS.24,DATA-BUS.25,DATA-BUS.26,DATA-BUS.27,DATA-BUS.28,
              DATA-BUS.29,DATA-BUS.30,DATA-BUS.31,I-DATA-IN.0,I-DATA-IN.1,
              I-DATA-IN.2,I-DATA-IN.3,I-DATA-IN.4,I-DATA-IN.5,I-DATA-IN.6,
              I-DATA-IN.7,I-DATA-IN.8,I-DATA-IN.9,I-DATA-IN.10,I-DATA-IN.11,
              I-DATA-IN.12,I-DATA-IN.13,I-DATA-IN.14,I-DATA-IN.15,
              I-DATA-IN.16,I-DATA-IN.17,I-DATA-IN.18,I-DATA-IN.19,
              I-DATA-IN.20,I-DATA-IN.21,I-DATA-IN.22,I-DATA-IN.23,
              I-DATA-IN.24,I-DATA-IN.25,I-DATA-IN.26,I-DATA-IN.27,
              I-DATA-IN.28,I-DATA-IN.29,I-DATA-IN.30,I-DATA-IN.31)
  = TTL-BIDIRECT-PADS_32(I-RW-,TEST-REGFILE-PO,DATA-BUS.0,DATA-BUS.1,
                         DATA-BUS.2,DATA-BUS.3,DATA-BUS.4,DATA-BUS.5,
```

```
                        DATA-BUS.6,DATA-BUS.7,DATA-BUS.8,DATA-BUS.9,
                        DATA-BUS.10,DATA-BUS.11,DATA-BUS.12,DATA-BUS.13,
                        DATA-BUS.14,DATA-BUS.15,DATA-BUS.16,DATA-BUS.17,
                        DATA-BUS.18,DATA-BUS.19,DATA-BUS.20,DATA-BUS.21,
                        DATA-BUS.22,DATA-BUS.23,DATA-BUS.24,DATA-BUS.25,
                        DATA-BUS.26,DATA-BUS.27,DATA-BUS.28,DATA-BUS.29,
                        DATA-BUS.30,DATA-BUS.31,I-DATA-OUT.0,I-DATA-OUT.1,
                        I-DATA-OUT.2,I-DATA-OUT.3,I-DATA-OUT.4,
                        I-DATA-OUT.5,I-DATA-OUT.6,I-DATA-OUT.7,
                        I-DATA-OUT.8,I-DATA-OUT.9,I-DATA-OUT.10,
                        I-DATA-OUT.11,I-DATA-OUT.12,I-DATA-OUT.13,
                        I-DATA-OUT.14,I-DATA-OUT.15,I-DATA-OUT.16,
                        I-DATA-OUT.17,I-DATA-OUT.18,I-DATA-OUT.19,
                        I-DATA-OUT.20,I-DATA-OUT.21,I-DATA-OUT.22,
                        I-DATA-OUT.23,I-DATA-OUT.24,I-DATA-OUT.25,
                        I-DATA-OUT.26,I-DATA-OUT.27,I-DATA-OUT.28,
                        I-DATA-OUT.29,I-DATA-OUT.30,I-DATA-OUT.31);
PC-REG-PADS(PC-REG-PO,I-PC-REG.0,I-PC-REG.1,I-PC-REG.2,I-PC-REG.3)
  = TTL-INPUT-PADS_4(DATA-BUS-PO,PC-REG-IN.0,PC-REG-IN.1,PC-REG-IN.2,
                     PC-REG-IN.3);
MONITOR(I-PO) = PROCMON(PC-REG-PO,CLK-PO,B-TRUE-P,B-TRUE-P);
PO-PAD(PO) = &B4(I-PO);
TO-PAD(TO) = &B8RP(I-TO);
TIMING-PAD(TIMING) = &B8(I-TIMING);
HDACK-PAD(HDACK-) = &B8(I-HDACK-);
RW-PAD(RW-) = &BT8(I-RW-,I-EN-ADDR-OUT-);
STROBE-PAD(STROBE-) = &BT8(I-STROBE-,I-EN-ADDR-OUT-);
ADDR-OUT-PADS(ADDR-OUT.0,ADDR-OUT.1,ADDR-OUT.2,ADDR-OUT.3,ADDR-OUT.4,
              ADDR-OUT.5,ADDR-OUT.6,ADDR-OUT.7,ADDR-OUT.8,ADDR-OUT.9,
              ADDR-OUT.10,ADDR-OUT.11,ADDR-OUT.12,ADDR-OUT.13,ADDR-OUT.14,
              ADDR-OUT.15,ADDR-OUT.16,ADDR-OUT.17,ADDR-OUT.18,ADDR-OUT.19,
              ADDR-OUT.20,ADDR-OUT.21,ADDR-OUT.22,ADDR-OUT.23,ADDR-OUT.24,
              ADDR-OUT.25,ADDR-OUT.26,ADDR-OUT.27,ADDR-OUT.28,ADDR-OUT.29,
              ADDR-OUT.30,ADDR-OUT.31)
  = TTL-TRI-OUTPUT-PADS_32(I-EN-ADDR-OUT-,I-ADDR-OUT.0,I-ADDR-OUT.1,
                           I-ADDR-OUT.2,I-ADDR-OUT.3,I-ADDR-OUT.4,
                           I-ADDR-OUT.5,I-ADDR-OUT.6,I-ADDR-OUT.7,
                           I-ADDR-OUT.8,I-ADDR-OUT.9,I-ADDR-OUT.10,
                           I-ADDR-OUT.11,I-ADDR-OUT.12,I-ADDR-OUT.13,
                           I-ADDR-OUT.14,I-ADDR-OUT.15,I-ADDR-OUT.16,
                           I-ADDR-OUT.17,I-ADDR-OUT.18,I-ADDR-OUT.19,
                           I-ADDR-OUT.20,I-ADDR-OUT.21,I-ADDR-OUT.22,
                           I-ADDR-OUT.23,I-ADDR-OUT.24,I-ADDR-OUT.25,
                           I-ADDR-OUT.26,I-ADDR-OUT.27,I-ADDR-OUT.28,
                           I-ADDR-OUT.29,I-ADDR-OUT.30,I-ADDR-OUT.31);
FLAGS-PADS(FLAGS.0,FLAGS.1,FLAGS.2,FLAGS.3)
  = TTL-OUTPUT-PADS_4(I-FLAGS.0,I-FLAGS.1,I-FLAGS.2,I-FLAGS.3);
CNTL-STATE-PADS(CNTL-STATE.0,CNTL-STATE.1,CNTL-STATE.2,CNTL-STATE.3,
                CNTL-STATE.4)
  = TTL-OUTPUT-PADS_5(I-CNTL-STATE.0,I-CNTL-STATE.1,I-CNTL-STATE.2,
                      I-CNTL-STATE.3,I-CNTL-STATE.4);
I-REG-PADS(I-REG.28,I-REG.29,I-REG.30,I-REG.31)
  = TTL-OUTPUT-PADS_4(I-I-REG.28,I-I-REG.29,I-I-REG.30,I-I-REG.31);
END MODULE;
```

```
MODULE TTL-BIDIRECT-PADS_32;
INPUTS ENABLE,PI,DATA.0,DATA.1,DATA.2,DATA.3,DATA.4,DATA.5,DATA.6,DATA.7,
       DATA.8,DATA.9,DATA.10,DATA.11,DATA.12,DATA.13,DATA.14,DATA.15,DATA.16,
       DATA.17,DATA.18,DATA.19,DATA.20,DATA.21,DATA.22,DATA.23,DATA.24,
       DATA.25,DATA.26,DATA.27,DATA.28,DATA.29,DATA.30,DATA.31,IN.0,IN.1,
       IN.2,IN.3,IN.4,IN.5,IN.6,IN.7,IN.8,IN.9,IN.10,IN.11,IN.12,IN.13,IN.14,
       IN.15,IN.16,IN.17,IN.18,IN.19,IN.20,IN.21,IN.22,IN.23,IN.24,IN.25,
       IN.26,IN.27,IN.28,IN.29,IN.30,IN.31;
OUTPUTS PO.31,DATA.0,DATA.1,DATA.2,DATA.3,DATA.4,DATA.5,DATA.6,DATA.7,DATA.8,
        DATA.9,DATA.10,DATA.11,DATA.12,DATA.13,DATA.14,DATA.15,DATA.16,
        DATA.17,DATA.18,DATA.19,DATA.20,DATA.21,DATA.22,DATA.23,DATA.24,
        DATA.25,DATA.26,DATA.27,DATA.28,DATA.29,DATA.30,DATA.31,OUT.0,OUT.1,
        OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7,OUT.8,OUT.9,OUT.10,OUT.11,OUT.12,
        OUT.13,OUT.14,OUT.15,OUT.16,OUT.17,OUT.18,OUT.19,OUT.20,OUT.21,
        OUT.22,OUT.23,OUT.24,OUT.25,OUT.26,OUT.27,OUT.28,OUT.29,OUT.30,
        OUT.31;
LEVEL FUNCTION;
DEFINE
ENABLE-BUF(BUF-ENABLE) = B-BUF-PWR(ENABLE);
G.0(DATA.0,OUT.0,PO.0) = &BD8TRP(DATA.0,IN.0,BUF-ENABLE,PI);
G.1(DATA.1,OUT.1,PO.1) = &BD8TRP(DATA.1,IN.1,BUF-ENABLE,PO.0);
G.2(DATA.2,OUT.2,PO.2) = &BD8TRP(DATA.2,IN.2,BUF-ENABLE,PO.1);
G.3(DATA.3,OUT.3,PO.3) = &BD8TRP(DATA.3,IN.3,BUF-ENABLE,PO.2);
G.4(DATA.4,OUT.4,PO.4) = &BD8TRP(DATA.4,IN.4,BUF-ENABLE,PO.3);
G.5(DATA.5,OUT.5,PO.5) = &BD8TRP(DATA.5,IN.5,BUF-ENABLE,PO.4);
G.6(DATA.6,OUT.6,PO.6) = &BD8TRP(DATA.6,IN.6,BUF-ENABLE,PO.5);
G.7(DATA.7,OUT.7,PO.7) = &BD8TRP(DATA.7,IN.7,BUF-ENABLE,PO.6);
G.8(DATA.8,OUT.8,PO.8) = &BD8TRP(DATA.8,IN.8,BUF-ENABLE,PO.7);
G.9(DATA.9,OUT.9,PO.9) = &BD8TRP(DATA.9,IN.9,BUF-ENABLE,PO.8);
G.10(DATA.10,OUT.10,PO.10) = &BD8TRP(DATA.10,IN.10,BUF-ENABLE,PO.9);
G.11(DATA.11,OUT.11,PO.11) = &BD8TRP(DATA.11,IN.11,BUF-ENABLE,PO.10);
G.12(DATA.12,OUT.12,PO.12) = &BD8TRP(DATA.12,IN.12,BUF-ENABLE,PO.11);
G.13(DATA.13,OUT.13,PO.13) = &BD8TRP(DATA.13,IN.13,BUF-ENABLE,PO.12);
G.14(DATA.14,OUT.14,PO.14) = &BD8TRP(DATA.14,IN.14,BUF-ENABLE,PO.13);
G.15(DATA.15,OUT.15,PO.15) = &BD8TRP(DATA.15,IN.15,BUF-ENABLE,PO.14);
G.16(DATA.16,OUT.16,PO.16) = &BD8TRP(DATA.16,IN.16,BUF-ENABLE,PO.15);
G.17(DATA.17,OUT.17,PO.17) = &BD8TRP(DATA.17,IN.17,BUF-ENABLE,PO.16);
G.18(DATA.18,OUT.18,PO.18) = &BD8TRP(DATA.18,IN.18,BUF-ENABLE,PO.17);
G.19(DATA.19,OUT.19,PO.19) = &BD8TRP(DATA.19,IN.19,BUF-ENABLE,PO.18);
G.20(DATA.20,OUT.20,PO.20) = &BD8TRP(DATA.20,IN.20,BUF-ENABLE,PO.19);
G.21(DATA.21,OUT.21,PO.21) = &BD8TRP(DATA.21,IN.21,BUF-ENABLE,PO.20);
G.22(DATA.22,OUT.22,PO.22) = &BD8TRP(DATA.22,IN.22,BUF-ENABLE,PO.21);
G.23(DATA.23,OUT.23,PO.23) = &BD8TRP(DATA.23,IN.23,BUF-ENABLE,PO.22);
G.24(DATA.24,OUT.24,PO.24) = &BD8TRP(DATA.24,IN.24,BUF-ENABLE,PO.23);
G.25(DATA.25,OUT.25,PO.25) = &BD8TRP(DATA.25,IN.25,BUF-ENABLE,PO.24);
G.26(DATA.26,OUT.26,PO.26) = &BD8TRP(DATA.26,IN.26,BUF-ENABLE,PO.25);
G.27(DATA.27,OUT.27,PO.27) = &BD8TRP(DATA.27,IN.27,BUF-ENABLE,PO.26);
G.28(DATA.28,OUT.28,PO.28) = &BD8TRP(DATA.28,IN.28,BUF-ENABLE,PO.27);
G.29(DATA.29,OUT.29,PO.29) = &BD8TRP(DATA.29,IN.29,BUF-ENABLE,PO.28);
G.30(DATA.30,OUT.30,PO.30) = &BD8TRP(DATA.30,IN.30,BUF-ENABLE,PO.29);
G.31(DATA.31,OUT.31,PO.31) = &BD8TRP(DATA.31,IN.31,BUF-ENABLE,PO.30);
END MODULE;

MODULE CHIP-MODULE;
INPUTS CLK,TI,TE,DTACK-,RESET-,HOLD-,DISABLE-REGFILE-,TEST-REGFILE-,
```

```
       PC-REG-IN.0,PC-REG-IN.1,PC-REG-IN.2,PC-REG-IN.3,DATA-IN.0,DATA-IN.1,
       DATA-IN.2,DATA-IN.3,DATA-IN.4,DATA-IN.5,DATA-IN.6,DATA-IN.7,DATA-IN.8,
       DATA-IN.9,DATA-IN.10,DATA-IN.11,DATA-IN.12,DATA-IN.13,DATA-IN.14,
       DATA-IN.15,DATA-IN.16,DATA-IN.17,DATA-IN.18,DATA-IN.19,DATA-IN.20,
       DATA-IN.21,DATA-IN.22,DATA-IN.23,DATA-IN.24,DATA-IN.25,DATA-IN.26,
       DATA-IN.27,DATA-IN.28,DATA-IN.29,DATA-IN.30,DATA-IN.31;
OUTPUTS TO,TIMING,HDACK-,EN-ADDR-OUT-,RW-,STROBE-,ADDR-OUT.0,ADDR-OUT.1,
        ADDR-OUT.2,ADDR-OUT.3,ADDR-OUT.4,ADDR-OUT.5,ADDR-OUT.6,ADDR-OUT.7,
        ADDR-OUT.8,ADDR-OUT.9,ADDR-OUT.10,ADDR-OUT.11,ADDR-OUT.12,
        ADDR-OUT.13,ADDR-OUT.14,ADDR-OUT.15,ADDR-OUT.16,ADDR-OUT.17,
        ADDR-OUT.18,ADDR-OUT.19,ADDR-OUT.20,ADDR-OUT.21,ADDR-OUT.22,
        ADDR-OUT.23,ADDR-OUT.24,ADDR-OUT.25,ADDR-OUT.26,ADDR-OUT.27,
        ADDR-OUT.28,ADDR-OUT.29,ADDR-OUT.30,ADDR-OUT.31,DATA-OUT.0,
        DATA-OUT.1,DATA-OUT.2,DATA-OUT.3,DATA-OUT.4,DATA-OUT.5,DATA-OUT.6,
        DATA-OUT.7,DATA-OUT.8,DATA-OUT.9,DATA-OUT.10,DATA-OUT.11,DATA-OUT.12,
        DATA-OUT.13,DATA-OUT.14,DATA-OUT.15,DATA-OUT.16,DATA-OUT.17,
        DATA-OUT.18,DATA-OUT.19,DATA-OUT.20,DATA-OUT.21,DATA-OUT.22,
        DATA-OUT.23,DATA-OUT.24,DATA-OUT.25,DATA-OUT.26,DATA-OUT.27,
        DATA-OUT.28,DATA-OUT.29,DATA-OUT.30,DATA-OUT.31,FLAGS.0,FLAGS.1,
        FLAGS.2,FLAGS.3,CNTL-STATE.0,CNTL-STATE.1,CNTL-STATE.2,CNTL-STATE.3,
        CNTL-STATE.4,I-REG.28,I-REG.29,I-REG.30,I-REG.31;
LEVEL FUNCTION;
DEFINE
CNTL-STATE(RW-SIG-,STROBE-,HDACK-,WE-REGS,WE-A-REG,WE-B-REG,WE-I-REG,
           WE-DATA-OUT,WE-ADDR-OUT,WE-HOLD-,WE-PC-REG,DATA-IN-SELECT,
           DEC-ADDR-OUT,SELECT-IMMEDIATE,ALU-C,ALU-ZERO,CNTL-STATE.0,
           CNTL-STATE.1,CNTL-STATE.2,CNTL-STATE.3,CNTL-STATE.4,WE-FLAGS.0,
           WE-FLAGS.1,WE-FLAGS.2,WE-FLAGS.3,REGS-ADDRESS.0,REGS-ADDRESS.1,
           REGS-ADDRESS.2,REGS-ADDRESS.3,ALU-OP.0,ALU-OP.1,ALU-OP.2,ALU-OP.3,
           ALU-MPG.0,ALU-MPG.1,ALU-MPG.2,ALU-MPG.3,ALU-MPG.4,ALU-MPG.5,
           ALU-MPG.6)
  = REG_40(CLK,TE-SIG,TI,NEXT-STATE.0,NEXT-STATE.1,NEXT-STATE.2,
           NEXT-STATE.3,NEXT-STATE.4,NEXT-STATE.5,NEXT-STATE.6,NEXT-STATE.7,
           NEXT-STATE.8,NEXT-STATE.9,NEXT-STATE.10,NEXT-STATE.11,
           NEXT-STATE.12,NEXT-STATE.13,NEXT-STATE.14,NEXT-STATE.15,
           NEXT-STATE.16,NEXT-STATE.17,NEXT-STATE.18,NEXT-STATE.19,
           NEXT-STATE.20,NEXT-STATE.21,NEXT-STATE.22,NEXT-STATE.23,
           NEXT-STATE.24,NEXT-STATE.25,NEXT-STATE.26,NEXT-STATE.27,
           NEXT-STATE.28,NEXT-STATE.29,NEXT-STATE.30,NEXT-STATE.31,
           NEXT-STATE.32,NEXT-STATE.33,NEXT-STATE.34,NEXT-STATE.35,
           NEXT-STATE.36,NEXT-STATE.37,NEXT-STATE.38,NEXT-STATE.39);
REGS(REGFILE-TO,REGFILE-OUT.0,REGFILE-OUT.1,REGFILE-OUT.2,REGFILE-OUT.3,
     REGFILE-OUT.4,REGFILE-OUT.5,REGFILE-OUT.6,REGFILE-OUT.7,REGFILE-OUT.8,
     REGFILE-OUT.9,REGFILE-OUT.10,REGFILE-OUT.11,REGFILE-OUT.12,
     REGFILE-OUT.13,REGFILE-OUT.14,REGFILE-OUT.15,REGFILE-OUT.16,
     REGFILE-OUT.17,REGFILE-OUT.18,REGFILE-OUT.19,REGFILE-OUT.20,
     REGFILE-OUT.21,REGFILE-OUT.22,REGFILE-OUT.23,REGFILE-OUT.24,
     REGFILE-OUT.25,REGFILE-OUT.26,REGFILE-OUT.27,REGFILE-OUT.28,
     REGFILE-OUT.29,REGFILE-OUT.30,REGFILE-OUT.31)
  = REGFILE(CLK,TE-SIG,ALU-MPG.6,WE-REGS,DISABLE-REGFILE-,TEST-REGFILE-,
            REGS-ADDRESS.0,REGS-ADDRESS.1,REGS-ADDRESS.2,REGS-ADDRESS.3,
            ALU-BUS.3,ALU-BUS.4,ALU-BUS.5,ALU-BUS.6,ALU-BUS.7,ALU-BUS.8,
            ALU-BUS.9,ALU-BUS.10,ALU-BUS.11,ALU-BUS.12,ALU-BUS.13,
            ALU-BUS.14,ALU-BUS.15,ALU-BUS.16,ALU-BUS.17,ALU-BUS.18,
            ALU-BUS.19,ALU-BUS.20,ALU-BUS.21,ALU-BUS.22,ALU-BUS.23,
```

```
            ALU-BUS.24,ALU-BUS.25,ALU-BUS.26,ALU-BUS.27,ALU-BUS.28,
            ALU-BUS.29,ALU-BUS.30,ALU-BUS.31,ALU-BUS.32,ALU-BUS.33,
            ALU-BUS.34);
CVNZ-FLAGS(FLAGS.0,FLAGS.1,FLAGS.2,FLAGS.3)
  = FLAGS(CLK,TE-SIG,REGFILE-TO,WE-FLAGS.0,WE-FLAGS.1,WE-FLAGS.2,WE-FLAGS.3,
          ALU-BUS.0,ALU-BUS.1,ALU-BUS.2,ALU-BUS.3,ALU-BUS.4,ALU-BUS.5,
          ALU-BUS.6,ALU-BUS.7,ALU-BUS.8,ALU-BUS.9,ALU-BUS.10,ALU-BUS.11,
          ALU-BUS.12,ALU-BUS.13,ALU-BUS.14,ALU-BUS.15,ALU-BUS.16,ALU-BUS.17,
          ALU-BUS.18,ALU-BUS.19,ALU-BUS.20,ALU-BUS.21,ALU-BUS.22,ALU-BUS.23,
          ALU-BUS.24,ALU-BUS.25,ALU-BUS.26,ALU-BUS.27,ALU-BUS.28,ALU-BUS.29,
          ALU-BUS.30,ALU-BUS.31,ALU-BUS.32,ALU-BUS.33,ALU-BUS.34);
A-REG(A-REG.0,A-REG.1,A-REG.2,A-REG.3,A-REG.4,A-REG.5,A-REG.6,A-REG.7,
      A-REG.8,A-REG.9,A-REG.10,A-REG.11,A-REG.12,A-REG.13,A-REG.14,A-REG.15,
      A-REG.16,A-REG.17,A-REG.18,A-REG.19,A-REG.20,A-REG.21,A-REG.22,
      A-REG.23,A-REG.24,A-REG.25,A-REG.26,A-REG.27,A-REG.28,A-REG.29,
      A-REG.30,A-REG.31)
  = WE-REG_32(CLK,WE-A-REG,TE-SIG,FLAGS.3,ABI-BUS.0,ABI-BUS.1,ABI-BUS.2,
              ABI-BUS.3,ABI-BUS.4,ABI-BUS.5,ABI-BUS.6,ABI-BUS.7,ABI-BUS.8,
              ABI-BUS.9,ABI-BUS.10,ABI-BUS.11,ABI-BUS.12,ABI-BUS.13,
              ABI-BUS.14,ABI-BUS.15,ABI-BUS.16,ABI-BUS.17,ABI-BUS.18,
              ABI-BUS.19,ABI-BUS.20,ABI-BUS.21,ABI-BUS.22,ABI-BUS.23,
              ABI-BUS.24,ABI-BUS.25,ABI-BUS.26,ABI-BUS.27,ABI-BUS.28,
              ABI-BUS.29,ABI-BUS.30,ABI-BUS.31);
B-REG(B-REG.0,B-REG.1,B-REG.2,B-REG.3,B-REG.4,B-REG.5,B-REG.6,B-REG.7,
      B-REG.8,B-REG.9,B-REG.10,B-REG.11,B-REG.12,B-REG.13,B-REG.14,B-REG.15,
      B-REG.16,B-REG.17,B-REG.18,B-REG.19,B-REG.20,B-REG.21,B-REG.22,
      B-REG.23,B-REG.24,B-REG.25,B-REG.26,B-REG.27,B-REG.28,B-REG.29,
      B-REG.30,B-REG.31)
  = WE-REG_32(CLK,WE-B-REG,TE-SIG,A-REG.31,ABI-BUS.0,ABI-BUS.1,ABI-BUS.2,
              ABI-BUS.3,ABI-BUS.4,ABI-BUS.5,ABI-BUS.6,ABI-BUS.7,ABI-BUS.8,
              ABI-BUS.9,ABI-BUS.10,ABI-BUS.11,ABI-BUS.12,ABI-BUS.13,
              ABI-BUS.14,ABI-BUS.15,ABI-BUS.16,ABI-BUS.17,ABI-BUS.18,
              ABI-BUS.19,ABI-BUS.20,ABI-BUS.21,ABI-BUS.22,ABI-BUS.23,
              ABI-BUS.24,ABI-BUS.25,ABI-BUS.26,ABI-BUS.27,ABI-BUS.28,
              ABI-BUS.29,ABI-BUS.30,ABI-BUS.31);
I-REG(I-REG.0,I-REG.1,I-REG.2,I-REG.3,I-REG.4,I-REG.5,I-REG.6,I-REG.7,
      I-REG.8,I-REG.9,I-REG.10,I-REG.11,I-REG.12,I-REG.13,I-REG.14,I-REG.15,
      I-REG.16,I-REG.17,I-REG.18,I-REG.19,I-REG.20,I-REG.21,I-REG.22,
      I-REG.23,I-REG.24,I-REG.25,I-REG.26,I-REG.27,I-REG.28,I-REG.29,
      I-REG.30,I-REG.31)
  = WE-REG_32(CLK,WE-I-REG,TE-SIG,B-REG.31,ABI-BUS.0,ABI-BUS.1,ABI-BUS.2,
              ABI-BUS.3,ABI-BUS.4,ABI-BUS.5,ABI-BUS.6,ABI-BUS.7,ABI-BUS.8,
              ABI-BUS.9,ABI-BUS.10,ABI-BUS.11,ABI-BUS.12,ABI-BUS.13,
              ABI-BUS.14,ABI-BUS.15,ABI-BUS.16,ABI-BUS.17,ABI-BUS.18,
              ABI-BUS.19,ABI-BUS.20,ABI-BUS.21,ABI-BUS.22,ABI-BUS.23,
              ABI-BUS.24,ABI-BUS.25,ABI-BUS.26,ABI-BUS.27,ABI-BUS.28,
              ABI-BUS.29,ABI-BUS.30,ABI-BUS.31);
DATA-OUT(DATA-OUT.0,DATA-OUT.1,DATA-OUT.2,DATA-OUT.3,DATA-OUT.4,DATA-OUT.5,
         DATA-OUT.6,DATA-OUT.7,DATA-OUT.8,DATA-OUT.9,DATA-OUT.10,DATA-OUT.11,
         DATA-OUT.12,DATA-OUT.13,DATA-OUT.14,DATA-OUT.15,DATA-OUT.16,
         DATA-OUT.17,DATA-OUT.18,DATA-OUT.19,DATA-OUT.20,DATA-OUT.21,
         DATA-OUT.22,DATA-OUT.23,DATA-OUT.24,DATA-OUT.25,DATA-OUT.26,
         DATA-OUT.27,DATA-OUT.28,DATA-OUT.29,DATA-OUT.30,DATA-OUT.31)
  = WE-REG_32(CLK,WE-DATA-OUT,TE-SIG,I-REG.31,ALU-BUS.3,ALU-BUS.4,ALU-BUS.5,
              ALU-BUS.6,ALU-BUS.7,ALU-BUS.8,ALU-BUS.9,ALU-BUS.10,ALU-BUS.11,
```

```
              ALU-BUS.12,ALU-BUS.13,ALU-BUS.14,ALU-BUS.15,ALU-BUS.16,
              ALU-BUS.17,ALU-BUS.18,ALU-BUS.19,ALU-BUS.20,ALU-BUS.21,
              ALU-BUS.22,ALU-BUS.23,ALU-BUS.24,ALU-BUS.25,ALU-BUS.26,
              ALU-BUS.27,ALU-BUS.28,ALU-BUS.29,ALU-BUS.30,ALU-BUS.31,
              ALU-BUS.32,ALU-BUS.33,ALU-BUS.34);
ADDR-OUT(ADDR-OUT.0,ADDR-OUT.1,ADDR-OUT.2,ADDR-OUT.3,ADDR-OUT.4,ADDR-OUT.5,
         ADDR-OUT.6,ADDR-OUT.7,ADDR-OUT.8,ADDR-OUT.9,ADDR-OUT.10,ADDR-OUT.11,
         ADDR-OUT.12,ADDR-OUT.13,ADDR-OUT.14,ADDR-OUT.15,ADDR-OUT.16,
         ADDR-OUT.17,ADDR-OUT.18,ADDR-OUT.19,ADDR-OUT.20,ADDR-OUT.21,
         ADDR-OUT.22,ADDR-OUT.23,ADDR-OUT.24,ADDR-OUT.25,ADDR-OUT.26,
         ADDR-OUT.27,ADDR-OUT.28,ADDR-OUT.29,ADDR-OUT.30,ADDR-OUT.31)
  = WE-REG_32(CLK,WE-ADDR-OUT,TE-SIG,DATA-OUT.31,ADDR-OUT-BUS.0,
              ADDR-OUT-BUS.1,ADDR-OUT-BUS.2,ADDR-OUT-BUS.3,ADDR-OUT-BUS.4,
              ADDR-OUT-BUS.5,ADDR-OUT-BUS.6,ADDR-OUT-BUS.7,ADDR-OUT-BUS.8,
              ADDR-OUT-BUS.9,ADDR-OUT-BUS.10,ADDR-OUT-BUS.11,
              ADDR-OUT-BUS.12,ADDR-OUT-BUS.13,ADDR-OUT-BUS.14,
              ADDR-OUT-BUS.15,ADDR-OUT-BUS.16,ADDR-OUT-BUS.17,
              ADDR-OUT-BUS.18,ADDR-OUT-BUS.19,ADDR-OUT-BUS.20,
              ADDR-OUT-BUS.21,ADDR-OUT-BUS.22,ADDR-OUT-BUS.23,
              ADDR-OUT-BUS.24,ADDR-OUT-BUS.25,ADDR-OUT-BUS.26,
              ADDR-OUT-BUS.27,ADDR-OUT-BUS.28,ADDR-OUT-BUS.29,
              ADDR-OUT-BUS.30,ADDR-OUT-BUS.31);
RESET-LATCH(LAST-RESET-,LAST-RESET-INV) = FD1S(RESET-,CLK,ADDR-OUT.31,TE-SIG);
DTACK--OR(DTACK--OR-STROBE-) = OR2(STROBE-,DTACK-);
DTACK-LATCH(LAST-DTACK-,LAST-DTACK-INV)
  = FD1S(DTACK--OR-STROBE-,CLK,LAST-RESET-,TE-SIG);
HOLD-LATCH(LAST-HOLD-,LAST-HOLD-INV)
  = FD1SLP(HOLD-,CLK,WE-HOLD-,LAST-DTACK-,TE-SIG);
PC-REG(PC-REG.0,PC-REG.1,PC-REG.2,PC-REG.3)
  = WE-REG_4(CLK,WE-PC-REG,TE-SIG,LAST-HOLD-,PC-REG-IN.0,PC-REG-IN.1,
             PC-REG-IN.2,PC-REG-IN.3);
IMMEDIATE-PASS(REG-BUS.0,REG-BUS.1,REG-BUS.2,REG-BUS.3,REG-BUS.4,REG-BUS.5,
               REG-BUS.6,REG-BUS.7,REG-BUS.8,REG-BUS.9,REG-BUS.10,REG-BUS.11,
               REG-BUS.12,REG-BUS.13,REG-BUS.14,REG-BUS.15,REG-BUS.16,
               REG-BUS.17,REG-BUS.18,REG-BUS.19,REG-BUS.20,REG-BUS.21,
               REG-BUS.22,REG-BUS.23,REG-BUS.24,REG-BUS.25,REG-BUS.26,
               REG-BUS.27,REG-BUS.28,REG-BUS.29,REG-BUS.30,REG-BUS.31)
  = EXTEND-IMMEDIATE(SELECT-IMMEDIATE,I-REG.0,I-REG.1,I-REG.2,I-REG.3,
                     I-REG.4,I-REG.5,I-REG.6,I-REG.7,I-REG.8,REGFILE-OUT.0,
                     REGFILE-OUT.1,REGFILE-OUT.2,REGFILE-OUT.3,
                     REGFILE-OUT.4,REGFILE-OUT.5,REGFILE-OUT.6,
                     REGFILE-OUT.7,REGFILE-OUT.8,REGFILE-OUT.9,
                     REGFILE-OUT.10,REGFILE-OUT.11,REGFILE-OUT.12,
                     REGFILE-OUT.13,REGFILE-OUT.14,REGFILE-OUT.15,
                     REGFILE-OUT.16,REGFILE-OUT.17,REGFILE-OUT.18,
                     REGFILE-OUT.19,REGFILE-OUT.20,REGFILE-OUT.21,
                     REGFILE-OUT.22,REGFILE-OUT.23,REGFILE-OUT.24,
                     REGFILE-OUT.25,REGFILE-OUT.26,REGFILE-OUT.27,
                     REGFILE-OUT.28,REGFILE-OUT.29,REGFILE-OUT.30,
                     REGFILE-OUT.31);
DEC-PASS(ADDR-OUT-BUS.0,ADDR-OUT-BUS.1,ADDR-OUT-BUS.2,ADDR-OUT-BUS.3,
         ADDR-OUT-BUS.4,ADDR-OUT-BUS.5,ADDR-OUT-BUS.6,ADDR-OUT-BUS.7,
         ADDR-OUT-BUS.8,ADDR-OUT-BUS.9,ADDR-OUT-BUS.10,ADDR-OUT-BUS.11,
         ADDR-OUT-BUS.12,ADDR-OUT-BUS.13,ADDR-OUT-BUS.14,ADDR-OUT-BUS.15,
         ADDR-OUT-BUS.16,ADDR-OUT-BUS.17,ADDR-OUT-BUS.18,ADDR-OUT-BUS.19,
```

```
        ADDR-OUT-BUS.20,ADDR-OUT-BUS.21,ADDR-OUT-BUS.22,ADDR-OUT-BUS.23,
        ADDR-OUT-BUS.24,ADDR-OUT-BUS.25,ADDR-OUT-BUS.26,ADDR-OUT-BUS.27,
        ADDR-OUT-BUS.28,ADDR-OUT-BUS.29,ADDR-OUT-BUS.30,ADDR-OUT-BUS.31)
  = DEC-PASS_32(DEC-ADDR-OUT,REG-BUS.0,REG-BUS.1,REG-BUS.2,REG-BUS.3,
               REG-BUS.4,REG-BUS.5,REG-BUS.6,REG-BUS.7,REG-BUS.8,REG-BUS.9,
               REG-BUS.10,REG-BUS.11,REG-BUS.12,REG-BUS.13,REG-BUS.14,
               REG-BUS.15,REG-BUS.16,REG-BUS.17,REG-BUS.18,REG-BUS.19,
               REG-BUS.20,REG-BUS.21,REG-BUS.22,REG-BUS.23,REG-BUS.24,
               REG-BUS.25,REG-BUS.26,REG-BUS.27,REG-BUS.28,REG-BUS.29,
               REG-BUS.30,REG-BUS.31);
MUX-CNTL(ABI-CNTL) = ND2(DATA-IN-SELECT,LAST-DTACK-INV);
DATA-IN-MUX(ABI-BUS.0,ABI-BUS.1,ABI-BUS.2,ABI-BUS.3,ABI-BUS.4,ABI-BUS.5,
            ABI-BUS.6,ABI-BUS.7,ABI-BUS.8,ABI-BUS.9,ABI-BUS.10,ABI-BUS.11,
            ABI-BUS.12,ABI-BUS.13,ABI-BUS.14,ABI-BUS.15,ABI-BUS.16,
            ABI-BUS.17,ABI-BUS.18,ABI-BUS.19,ABI-BUS.20,ABI-BUS.21,
            ABI-BUS.22,ABI-BUS.23,ABI-BUS.24,ABI-BUS.25,ABI-BUS.26,
            ABI-BUS.27,ABI-BUS.28,ABI-BUS.29,ABI-BUS.30,ABI-BUS.31)
  = TV-IF_32(ABI-CNTL,REG-BUS.0,REG-BUS.1,REG-BUS.2,REG-BUS.3,REG-BUS.4,
             REG-BUS.5,REG-BUS.6,REG-BUS.7,REG-BUS.8,REG-BUS.9,REG-BUS.10,
             REG-BUS.11,REG-BUS.12,REG-BUS.13,REG-BUS.14,REG-BUS.15,
             REG-BUS.16,REG-BUS.17,REG-BUS.18,REG-BUS.19,REG-BUS.20,
             REG-BUS.21,REG-BUS.22,REG-BUS.23,REG-BUS.24,REG-BUS.25,
             REG-BUS.26,REG-BUS.27,REG-BUS.28,REG-BUS.29,REG-BUS.30,
             REG-BUS.31,DATA-IN.0,DATA-IN.1,DATA-IN.2,DATA-IN.3,DATA-IN.4,
             DATA-IN.5,DATA-IN.6,DATA-IN.7,DATA-IN.8,DATA-IN.9,DATA-IN.10,
             DATA-IN.11,DATA-IN.12,DATA-IN.13,DATA-IN.14,DATA-IN.15,
             DATA-IN.16,DATA-IN.17,DATA-IN.18,DATA-IN.19,DATA-IN.20,
             DATA-IN.21,DATA-IN.22,DATA-IN.23,DATA-IN.24,DATA-IN.25,
             DATA-IN.26,DATA-IN.27,DATA-IN.28,DATA-IN.29,DATA-IN.30,
             DATA-IN.31);
ALU(ALU-BUS.0,ALU-BUS.1,ALU-BUS.2,ALU-BUS.3,ALU-BUS.4,ALU-BUS.5,ALU-BUS.6,
    ALU-BUS.7,ALU-BUS.8,ALU-BUS.9,ALU-BUS.10,ALU-BUS.11,ALU-BUS.12,
    ALU-BUS.13,ALU-BUS.14,ALU-BUS.15,ALU-BUS.16,ALU-BUS.17,ALU-BUS.18,
    ALU-BUS.19,ALU-BUS.20,ALU-BUS.21,ALU-BUS.22,ALU-BUS.23,ALU-BUS.24,
    ALU-BUS.25,ALU-BUS.26,ALU-BUS.27,ALU-BUS.28,ALU-BUS.29,ALU-BUS.30,
    ALU-BUS.31,ALU-BUS.32,ALU-BUS.33,ALU-BUS.34)
  = CORE-ALU_32(ALU-C,A-REG.0,A-REG.1,A-REG.2,A-REG.3,A-REG.4,A-REG.5,
               A-REG.6,A-REG.7,A-REG.8,A-REG.9,A-REG.10,A-REG.11,A-REG.12,
               A-REG.13,A-REG.14,A-REG.15,A-REG.16,A-REG.17,A-REG.18,
               A-REG.19,A-REG.20,A-REG.21,A-REG.22,A-REG.23,A-REG.24,
               A-REG.25,A-REG.26,A-REG.27,A-REG.28,A-REG.29,A-REG.30,
               A-REG.31,B-REG.0,B-REG.1,B-REG.2,B-REG.3,B-REG.4,B-REG.5,
               B-REG.6,B-REG.7,B-REG.8,B-REG.9,B-REG.10,B-REG.11,B-REG.12,
               B-REG.13,B-REG.14,B-REG.15,B-REG.16,B-REG.17,B-REG.18,
               B-REG.19,B-REG.20,B-REG.21,B-REG.22,B-REG.23,B-REG.24,
               B-REG.25,B-REG.26,B-REG.27,B-REG.28,B-REG.29,B-REG.30,
               B-REG.31,ALU-ZERO,ALU-MPG.0,ALU-MPG.1,ALU-MPG.2,ALU-MPG.3,
               ALU-MPG.4,ALU-MPG.5,ALU-MPG.6,ALU-OP.0,ALU-OP.1,ALU-OP.2,
               ALU-OP.3);
NEXT-STATE(NEXT-STATE.0,NEXT-STATE.1,NEXT-STATE.2,NEXT-STATE.3,NEXT-STATE.4,
           NEXT-STATE.5,NEXT-STATE.6,NEXT-STATE.7,NEXT-STATE.8,NEXT-STATE.9,
           NEXT-STATE.10,NEXT-STATE.11,NEXT-STATE.12,NEXT-STATE.13,
           NEXT-STATE.14,NEXT-STATE.15,NEXT-STATE.16,NEXT-STATE.17,
           NEXT-STATE.18,NEXT-STATE.19,NEXT-STATE.20,NEXT-STATE.21,
           NEXT-STATE.22,NEXT-STATE.23,NEXT-STATE.24,NEXT-STATE.25,
```

```
           NEXT-STATE.26,NEXT-STATE.27,NEXT-STATE.28,NEXT-STATE.29,
           NEXT-STATE.30,NEXT-STATE.31,NEXT-STATE.32,NEXT-STATE.33,
           NEXT-STATE.34,NEXT-STATE.35,NEXT-STATE.36,NEXT-STATE.37,
           NEXT-STATE.38,NEXT-STATE.39)
  = NEXT-CNTL-STATE(LAST-RESET-,LAST-DTACK-,LAST-HOLD-,RW-SIG-,CNTL-STATE.0,
                    CNTL-STATE.1,CNTL-STATE.2,CNTL-STATE.3,CNTL-STATE.4,
                    I-REG.0,I-REG.1,I-REG.2,I-REG.3,I-REG.4,I-REG.5,I-REG.6,
                    I-REG.7,I-REG.8,I-REG.9,I-REG.10,I-REG.11,I-REG.12,
                    I-REG.13,I-REG.14,I-REG.15,I-REG.16,I-REG.17,I-REG.18,
                    I-REG.19,I-REG.20,I-REG.21,I-REG.22,I-REG.23,I-REG.24,
                    I-REG.25,I-REG.26,I-REG.27,I-REG.28,I-REG.29,I-REG.30,
                    I-REG.31,FLAGS.0,FLAGS.1,FLAGS.2,FLAGS.3,PC-REG.0,
                    PC-REG.1,PC-REG.2,PC-REG.3,REGS-ADDRESS.0,
                    REGS-ADDRESS.1,REGS-ADDRESS.2,REGS-ADDRESS.3);
TE-BUFFER(TE-SIG) = B-BUF-PWR(TE);
RW-BUFFER(RW-) = B-BUF(RW-SIG-);
EN-ADDR-OUT-GATE(EN-ADDR-OUT-) = IVA(HDACK-);
TIMING-GATE(TIMING) = ID(ALU-BUS.2);
SCANOUT(TO) = ID(PC-REG.3);
END MODULE;

MODULE REGFILE;
INPUTS CLK,TE,TI,WE,DISABLE-REGFILE-,TEST-REGFILE-,ADDRESS.0,ADDRESS.1,
       ADDRESS.2,ADDRESS.3,DATA.0,DATA.1,DATA.2,DATA.3,DATA.4,DATA.5,DATA.6,
       DATA.7,DATA.8,DATA.9,DATA.10,DATA.11,DATA.12,DATA.13,DATA.14,DATA.15,
       DATA.16,DATA.17,DATA.18,DATA.19,DATA.20,DATA.21,DATA.22,DATA.23,
       DATA.24,DATA.25,DATA.26,DATA.27,DATA.28,DATA.29,DATA.30,DATA.31;
OUTPUTS TO,OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7,OUT.8,OUT.9,
        OUT.10,OUT.11,OUT.12,OUT.13,OUT.14,OUT.15,OUT.16,OUT.17,OUT.18,
        OUT.19,OUT.20,OUT.21,OUT.22,OUT.23,OUT.24,OUT.25,OUT.26,OUT.27,
        OUT.28,OUT.29,OUT.30,OUT.31;
LEVEL FUNCTION;
DEFINE
WE-LATCH(WE-DP-RAM,WE-DP-RAM-) = FD1S(WE,CLK,TI,TE);
ADDRESS-LATCH(ADDRESS-DP-RAM.0,ADDRESS-DP-RAM.1,ADDRESS-DP-RAM.2,
              ADDRESS-DP-RAM.3)
  = WE-REG_4(CLK,WE,TE,WE-DP-RAM,ADDRESS.0,ADDRESS.1,ADDRESS.2,ADDRESS.3);
DATA-LATCH(DATA-DP-RAM.0,DATA-DP-RAM.1,DATA-DP-RAM.2,DATA-DP-RAM.3,
           DATA-DP-RAM.4,DATA-DP-RAM.5,DATA-DP-RAM.6,DATA-DP-RAM.7,
           DATA-DP-RAM.8,DATA-DP-RAM.9,DATA-DP-RAM.10,DATA-DP-RAM.11,
           DATA-DP-RAM.12,DATA-DP-RAM.13,DATA-DP-RAM.14,DATA-DP-RAM.15,
           DATA-DP-RAM.16,DATA-DP-RAM.17,DATA-DP-RAM.18,DATA-DP-RAM.19,
           DATA-DP-RAM.20,DATA-DP-RAM.21,DATA-DP-RAM.22,DATA-DP-RAM.23,
           DATA-DP-RAM.24,DATA-DP-RAM.25,DATA-DP-RAM.26,DATA-DP-RAM.27,
           DATA-DP-RAM.28,DATA-DP-RAM.29,DATA-DP-RAM.30,DATA-DP-RAM.31)
  = WE-REG_32(CLK,WE,TE,ADDRESS-DP-RAM.3,DATA.0,DATA.1,DATA.2,DATA.3,DATA.4,
              DATA.5,DATA.6,DATA.7,DATA.8,DATA.9,DATA.10,DATA.11,DATA.12,
              DATA.13,DATA.14,DATA.15,DATA.16,DATA.17,DATA.18,DATA.19,
              DATA.20,DATA.21,DATA.22,DATA.23,DATA.24,DATA.25,DATA.26,
              DATA.27,DATA.28,DATA.29,DATA.30,DATA.31);
REG-EN-CIRCUIT(WE-)
  = RAM-ENABLE-CIRCUIT(CLK,TEST-REGFILE-,DISABLE-REGFILE-,WE-DP-RAM);
RAM(RAMOUT.0,RAMOUT.1,RAMOUT.2,RAMOUT.3,RAMOUT.4,RAMOUT.5,RAMOUT.6,RAMOUT.7,
    RAMOUT.8,RAMOUT.9,RAMOUT.10,RAMOUT.11,RAMOUT.12,RAMOUT.13,RAMOUT.14,
    RAMOUT.15,RAMOUT.16,RAMOUT.17,RAMOUT.18,RAMOUT.19,RAMOUT.20,RAMOUT.21,
```

```
    RAMOUT.22,RAMOUT.23,RAMOUT.24,RAMOUT.25,RAMOUT.26,RAMOUT.27,RAMOUT.28,
    RAMOUT.29,RAMOUT.30,RAMOUT.31)
  = CMRB100A(ADDRESS.0,ADDRESS.1,ADDRESS.2,ADDRESS.3,ADDRESS-DP-RAM.0,
             ADDRESS-DP-RAM.1,ADDRESS-DP-RAM.2,ADDRESS-DP-RAM.3,WE-,
             DATA-DP-RAM.0,DATA-DP-RAM.1,DATA-DP-RAM.2,DATA-DP-RAM.3,
             DATA-DP-RAM.4,DATA-DP-RAM.5,DATA-DP-RAM.6,DATA-DP-RAM.7,
             DATA-DP-RAM.8,DATA-DP-RAM.9,DATA-DP-RAM.10,DATA-DP-RAM.11,
             DATA-DP-RAM.12,DATA-DP-RAM.13,DATA-DP-RAM.14,DATA-DP-RAM.15,
             DATA-DP-RAM.16,DATA-DP-RAM.17,DATA-DP-RAM.18,DATA-DP-RAM.19,
             DATA-DP-RAM.20,DATA-DP-RAM.21,DATA-DP-RAM.22,DATA-DP-RAM.23,
             DATA-DP-RAM.24,DATA-DP-RAM.25,DATA-DP-RAM.26,DATA-DP-RAM.27,
             DATA-DP-RAM.28,DATA-DP-RAM.29,DATA-DP-RAM.30,DATA-DP-RAM.31);
COMPARE(READ-EQUAL-WRITE)
  = V-EQUAL_4(ADDRESS.0,ADDRESS.1,ADDRESS.2,ADDRESS.3,ADDRESS-DP-RAM.0,
              ADDRESS-DP-RAM.1,ADDRESS-DP-RAM.2,ADDRESS-DP-RAM.3);
MUX-CONTROL(S) = AN3(WE-DP-RAM,READ-EQUAL-WRITE,TEST-REGFILE-);
MUX(OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7,OUT.8,OUT.9,OUT.10,
    OUT.11,OUT.12,OUT.13,OUT.14,OUT.15,OUT.16,OUT.17,OUT.18,OUT.19,OUT.20,
    OUT.21,OUT.22,OUT.23,OUT.24,OUT.25,OUT.26,OUT.27,OUT.28,OUT.29,OUT.30,
    OUT.31)
  = TV-IF_32(S,DATA-DP-RAM.0,DATA-DP-RAM.1,DATA-DP-RAM.2,DATA-DP-RAM.3,
             DATA-DP-RAM.4,DATA-DP-RAM.5,DATA-DP-RAM.6,DATA-DP-RAM.7,
             DATA-DP-RAM.8,DATA-DP-RAM.9,DATA-DP-RAM.10,DATA-DP-RAM.11,
             DATA-DP-RAM.12,DATA-DP-RAM.13,DATA-DP-RAM.14,DATA-DP-RAM.15,
             DATA-DP-RAM.16,DATA-DP-RAM.17,DATA-DP-RAM.18,DATA-DP-RAM.19,
             DATA-DP-RAM.20,DATA-DP-RAM.21,DATA-DP-RAM.22,DATA-DP-RAM.23,
             DATA-DP-RAM.24,DATA-DP-RAM.25,DATA-DP-RAM.26,DATA-DP-RAM.27,
             DATA-DP-RAM.28,DATA-DP-RAM.29,DATA-DP-RAM.30,DATA-DP-RAM.31,
             RAMOUT.0,RAMOUT.1,RAMOUT.2,RAMOUT.3,RAMOUT.4,RAMOUT.5,RAMOUT.6,
             RAMOUT.7,RAMOUT.8,RAMOUT.9,RAMOUT.10,RAMOUT.11,RAMOUT.12,
             RAMOUT.13,RAMOUT.14,RAMOUT.15,RAMOUT.16,RAMOUT.17,RAMOUT.18,
             RAMOUT.19,RAMOUT.20,RAMOUT.21,RAMOUT.22,RAMOUT.23,RAMOUT.24,
             RAMOUT.25,RAMOUT.26,RAMOUT.27,RAMOUT.28,RAMOUT.29,RAMOUT.30,
             RAMOUT.31);
SCANOUT(TO) = ID(DATA-DP-RAM.31);
END MODULE;

MODULE V-EQUAL_4;
INPUTS A.0,A.1,A.2,A.3,B.0,B.1,B.2,B.3;
OUTPUTS EQUAL;
LEVEL FUNCTION;
DEFINE
G0(X.0,X.1,X.2,X.3) = V-XOR_4(A.0,A.1,A.2,A.3,B.0,B.1,B.2,B.3);
G1(EQUAL) = TV-ZEROP_4(X.0,X.1,X.2,X.3);
END MODULE;

MODULE V-XOR_4;
INPUTS A.0,A.1,A.2,A.3,B.0,B.1,B.2,B.3;
OUTPUTS Y.0,Y.1,Y.2,Y.3;
LEVEL FUNCTION;
DEFINE
G.0(Y.0) = EO(A.0,B.0);
G.1(Y.1) = EO(A.1,B.1);
G.2(Y.2) = EO(A.2,B.2);
G.3(Y.3) = EO(A.3,B.3);
```

```
END MODULE;

MODULE TV-ZEROP_4;
INPUTS IN.0,IN.1,IN.2,IN.3;
OUTPUTS OUT;
LEVEL FUNCTION;
DEFINE
G0(OUT-) = T-OR_4(IN.0,IN.1,IN.2,IN.3);
G1(OUT) = IVA(OUT-);
END MODULE;

MODULE T-OR_4;
INPUTS A.0,A.1,A.2,A.3;
OUTPUTS OUT;
LEVEL FUNCTION;
DEFINE
LEFT(LEFT-OUT) = T-NOR_2(A.0,A.1);
RIGHT(RIGHT-OUT) = T-NOR_2(A.2,A.3);
OUTPUT(OUT) = ND2(LEFT-OUT,RIGHT-OUT);
END MODULE;

MODULE T-NOR_2;
INPUTS A.0,A.1;
OUTPUTS OUT;
LEVEL FUNCTION;
DEFINE
LEAF(OUT) = NR2(A.0,A.1);
END MODULE;

MODULE FLAGS;
INPUTS CLK,TE,TI,SET-FLAGS.0,SET-FLAGS.1,SET-FLAGS.2,SET-FLAGS.3,CVZBV.0,
       CVZBV.1,CVZBV.2,CVZBV.3,CVZBV.4,CVZBV.5,CVZBV.6,CVZBV.7,CVZBV.8,
       CVZBV.9,CVZBV.10,CVZBV.11,CVZBV.12,CVZBV.13,CVZBV.14,CVZBV.15,
       CVZBV.16,CVZBV.17,CVZBV.18,CVZBV.19,CVZBV.20,CVZBV.21,CVZBV.22,
       CVZBV.23,CVZBV.24,CVZBV.25,CVZBV.26,CVZBV.27,CVZBV.28,CVZBV.29,
       CVZBV.30,CVZBV.31,CVZBV.32,CVZBV.33,CVZBV.34;
OUTPUTS Z,N,V,C;
LEVEL FUNCTION;
DEFINE
Z-LATCH(Z,ZB) = FD1SLP(CVZBV.2,CLK,SET-FLAGS.0,TI,TE);
N-LATCH(N,NB) = FD1SLP(CVZBV.34,CLK,SET-FLAGS.1,Z,TE);
V-LATCH(V,VB) = FD1SLP(CVZBV.1,CLK,SET-FLAGS.2,N,TE);
C-LATCH(C,CB) = FD1SLP(CVZBV.0,CLK,SET-FLAGS.3,V,TE);
END MODULE;

MODULE WE-REG_32;
INPUTS CLK,WE,TE,TI,D.0,D.1,D.2,D.3,D.4,D.5,D.6,D.7,D.8,D.9,D.10,D.11,D.12,
       D.13,D.14,D.15,D.16,D.17,D.18,D.19,D.20,D.21,D.22,D.23,D.24,D.25,D.26,
       D.27,D.28,D.29,D.30,D.31;
OUTPUTS Q.0,Q.1,Q.2,Q.3,Q.4,Q.5,Q.6,Q.7,Q.8,Q.9,Q.10,Q.11,Q.12,Q.13,Q.14,
        Q.15,Q.16,Q.17,Q.18,Q.19,Q.20,Q.21,Q.22,Q.23,Q.24,Q.25,Q.26,Q.27,
        Q.28,Q.29,Q.30,Q.31;
LEVEL FUNCTION;
DEFINE
WE-BUFFER(WE-BUF) = B-BUF-PWR(WE);
```

```
TE-BUFFER(TE-BUF) = B-BUF-PWR(TE);
TI-DEL(TI-BUF) = DEL4(TI);
G.0(Q.0,QB.0) = FD1SLP(D.0,CLK,WE-BUF,TI-BUF,TE-BUF);
G.1(Q.1,QB.1) = FD1SLP(D.1,CLK,WE-BUF,Q.0,TE-BUF);
G.2(Q.2,QB.2) = FD1SLP(D.2,CLK,WE-BUF,Q.1,TE-BUF);
G.3(Q.3,QB.3) = FD1SLP(D.3,CLK,WE-BUF,Q.2,TE-BUF);
G.4(Q.4,QB.4) = FD1SLP(D.4,CLK,WE-BUF,Q.3,TE-BUF);
G.5(Q.5,QB.5) = FD1SLP(D.5,CLK,WE-BUF,Q.4,TE-BUF);
G.6(Q.6,QB.6) = FD1SLP(D.6,CLK,WE-BUF,Q.5,TE-BUF);
G.7(Q.7,QB.7) = FD1SLP(D.7,CLK,WE-BUF,Q.6,TE-BUF);
G.8(Q.8,QB.8) = FD1SLP(D.8,CLK,WE-BUF,Q.7,TE-BUF);
G.9(Q.9,QB.9) = FD1SLP(D.9,CLK,WE-BUF,Q.8,TE-BUF);
G.10(Q.10,QB.10) = FD1SLP(D.10,CLK,WE-BUF,Q.9,TE-BUF);
G.11(Q.11,QB.11) = FD1SLP(D.11,CLK,WE-BUF,Q.10,TE-BUF);
G.12(Q.12,QB.12) = FD1SLP(D.12,CLK,WE-BUF,Q.11,TE-BUF);
G.13(Q.13,QB.13) = FD1SLP(D.13,CLK,WE-BUF,Q.12,TE-BUF);
G.14(Q.14,QB.14) = FD1SLP(D.14,CLK,WE-BUF,Q.13,TE-BUF);
G.15(Q.15,QB.15) = FD1SLP(D.15,CLK,WE-BUF,Q.14,TE-BUF);
G.16(Q.16,QB.16) = FD1SLP(D.16,CLK,WE-BUF,Q.15,TE-BUF);
G.17(Q.17,QB.17) = FD1SLP(D.17,CLK,WE-BUF,Q.16,TE-BUF);
G.18(Q.18,QB.18) = FD1SLP(D.18,CLK,WE-BUF,Q.17,TE-BUF);
G.19(Q.19,QB.19) = FD1SLP(D.19,CLK,WE-BUF,Q.18,TE-BUF);
G.20(Q.20,QB.20) = FD1SLP(D.20,CLK,WE-BUF,Q.19,TE-BUF);
G.21(Q.21,QB.21) = FD1SLP(D.21,CLK,WE-BUF,Q.20,TE-BUF);
G.22(Q.22,QB.22) = FD1SLP(D.22,CLK,WE-BUF,Q.21,TE-BUF);
G.23(Q.23,QB.23) = FD1SLP(D.23,CLK,WE-BUF,Q.22,TE-BUF);
G.24(Q.24,QB.24) = FD1SLP(D.24,CLK,WE-BUF,Q.23,TE-BUF);
G.25(Q.25,QB.25) = FD1SLP(D.25,CLK,WE-BUF,Q.24,TE-BUF);
G.26(Q.26,QB.26) = FD1SLP(D.26,CLK,WE-BUF,Q.25,TE-BUF);
G.27(Q.27,QB.27) = FD1SLP(D.27,CLK,WE-BUF,Q.26,TE-BUF);
G.28(Q.28,QB.28) = FD1SLP(D.28,CLK,WE-BUF,Q.27,TE-BUF);
G.29(Q.29,QB.29) = FD1SLP(D.29,CLK,WE-BUF,Q.28,TE-BUF);
G.30(Q.30,QB.30) = FD1SLP(D.30,CLK,WE-BUF,Q.29,TE-BUF);
G.31(Q.31,QB.31) = FD1SLP(D.31,CLK,WE-BUF,Q.30,TE-BUF);
END MODULE;

MODULE WE-REG_4;
INPUTS CLK,WE,TE,TI,D.0,D.1,D.2,D.3;
OUTPUTS Q.0,Q.1,Q.2,Q.3;
LEVEL FUNCTION;
DEFINE
WE-BUFFER(WE-BUF) = B-BUF(WE);
TE-BUFFER(TE-BUF) = B-BUF(TE);
TI-DEL(TI-BUF) = DEL4(TI);
G.0(Q.0,QB.0) = FD1SLP(D.0,CLK,WE-BUF,TI-BUF,TE-BUF);
G.1(Q.1,QB.1) = FD1SLP(D.1,CLK,WE-BUF,Q.0,TE-BUF);
G.2(Q.2,QB.2) = FD1SLP(D.2,CLK,WE-BUF,Q.1,TE-BUF);
G.3(Q.3,QB.3) = FD1SLP(D.3,CLK,WE-BUF,Q.2,TE-BUF);
END MODULE;

MODULE REG_40;
INPUTS CLK,TE,TI,D.0,D.1,D.2,D.3,D.4,D.5,D.6,D.7,D.8,D.9,D.10,D.11,D.12,D.13,
       D.14,D.15,D.16,D.17,D.18,D.19,D.20,D.21,D.22,D.23,D.24,D.25,D.26,D.27,
       D.28,D.29,D.30,D.31,D.32,D.33,D.34,D.35,D.36,D.37,D.38,D.39;
OUTPUTS Q.0,Q.1,Q.2,Q.3,Q.4,Q.5,Q.6,Q.7,Q.8,Q.9,Q.10,Q.11,Q.12,Q.13,Q.14,
```

```
        Q.15,Q.16,Q.17,Q.18,Q.19,Q.20,Q.21,Q.22,Q.23,Q.24,Q.25,Q.26,Q.27,
        Q.28,Q.29,Q.30,Q.31,Q.32,Q.33,Q.34,Q.35,Q.36,Q.37,Q.38,Q.39;
LEVEL FUNCTION;
DEFINE
TE-BUFFER(TE-BUF) = B-BUF-PWR(TE);
TI-DEL(TI-BUF) = DEL4(TI);
G.0(Q.0,QB.0) = FD1S(D.0,CLK,TI-BUF,TE-BUF);
G.1(Q.1,QB.1) = FD1S(D.1,CLK,Q.0,TE-BUF);
G.2(Q.2,QB.2) = FD1S(D.2,CLK,Q.1,TE-BUF);
G.3(Q.3,QB.3) = FD1S(D.3,CLK,Q.2,TE-BUF);
G.4(Q.4,QB.4) = FD1S(D.4,CLK,Q.3,TE-BUF);
G.5(Q.5,QB.5) = FD1S(D.5,CLK,Q.4,TE-BUF);
G.6(Q.6,QB.6) = FD1S(D.6,CLK,Q.5,TE-BUF);
G.7(Q.7,QB.7) = FD1S(D.7,CLK,Q.6,TE-BUF);
G.8(Q.8,QB.8) = FD1S(D.8,CLK,Q.7,TE-BUF);
G.9(Q.9,QB.9) = FD1S(D.9,CLK,Q.8,TE-BUF);
G.10(Q.10,QB.10) = FD1S(D.10,CLK,Q.9,TE-BUF);
G.11(Q.11,QB.11) = FD1S(D.11,CLK,Q.10,TE-BUF);
G.12(Q.12,QB.12) = FD1S(D.12,CLK,Q.11,TE-BUF);
G.13(Q.13,QB.13) = FD1S(D.13,CLK,Q.12,TE-BUF);
G.14(Q.14,QB.14) = FD1S(D.14,CLK,Q.13,TE-BUF);
G.15(Q.15,QB.15) = FD1S(D.15,CLK,Q.14,TE-BUF);
G.16(Q.16,QB.16) = FD1S(D.16,CLK,Q.15,TE-BUF);
G.17(Q.17,QB.17) = FD1S(D.17,CLK,Q.16,TE-BUF);
G.18(Q.18,QB.18) = FD1S(D.18,CLK,Q.17,TE-BUF);
G.19(Q.19,QB.19) = FD1S(D.19,CLK,Q.18,TE-BUF);
G.20(Q.20,QB.20) = FD1S(D.20,CLK,Q.19,TE-BUF);
G.21(Q.21,QB.21) = FD1S(D.21,CLK,Q.20,TE-BUF);
G.22(Q.22,QB.22) = FD1S(D.22,CLK,Q.21,TE-BUF);
G.23(Q.23,QB.23) = FD1S(D.23,CLK,Q.22,TE-BUF);
G.24(Q.24,QB.24) = FD1S(D.24,CLK,Q.23,TE-BUF);
G.25(Q.25,QB.25) = FD1S(D.25,CLK,Q.24,TE-BUF);
G.26(Q.26,QB.26) = FD1S(D.26,CLK,Q.25,TE-BUF);
G.27(Q.27,QB.27) = FD1S(D.27,CLK,Q.26,TE-BUF);
G.28(Q.28,QB.28) = FD1S(D.28,CLK,Q.27,TE-BUF);
G.29(Q.29,QB.29) = FD1S(D.29,CLK,Q.28,TE-BUF);
G.30(Q.30,QB.30) = FD1S(D.30,CLK,Q.29,TE-BUF);
G.31(Q.31,QB.31) = FD1S(D.31,CLK,Q.30,TE-BUF);
G.32(Q.32,QB.32) = FD1S(D.32,CLK,Q.31,TE-BUF);
G.33(Q.33,QB.33) = FD1S(D.33,CLK,Q.32,TE-BUF);
G.34(Q.34,QB.34) = FD1S(D.34,CLK,Q.33,TE-BUF);
G.35(Q.35,QB.35) = FD1S(D.35,CLK,Q.34,TE-BUF);
G.36(Q.36,QB.36) = FD1S(D.36,CLK,Q.35,TE-BUF);
G.37(Q.37,QB.37) = FD1S(D.37,CLK,Q.36,TE-BUF);
G.38(Q.38,QB.38) = FD1S(D.38,CLK,Q.37,TE-BUF);
G.39(Q.39,QB.39) = FD1S(D.39,CLK,Q.38,TE-BUF);
END MODULE;

MODULE EXTEND-IMMEDIATE;
INPUTS SELECT-IMMEDIATE,IMMEDIATE.0,IMMEDIATE.1,IMMEDIATE.2,IMMEDIATE.3,
       IMMEDIATE.4,IMMEDIATE.5,IMMEDIATE.6,IMMEDIATE.7,IMMEDIATE.8,
       REG-DATA.0,REG-DATA.1,REG-DATA.2,REG-DATA.3,REG-DATA.4,REG-DATA.5,
       REG-DATA.6,REG-DATA.7,REG-DATA.8,REG-DATA.9,REG-DATA.10,REG-DATA.11,
       REG-DATA.12,REG-DATA.13,REG-DATA.14,REG-DATA.15,REG-DATA.16,
       REG-DATA.17,REG-DATA.18,REG-DATA.19,REG-DATA.20,REG-DATA.21,
```

```
       REG-DATA.22,REG-DATA.23,REG-DATA.24,REG-DATA.25,REG-DATA.26,
       REG-DATA.27,REG-DATA.28,REG-DATA.29,REG-DATA.30,REG-DATA.31;
OUTPUTS Z.0,Z.1,Z.2,Z.3,Z.4,Z.5,Z.6,Z.7,Z.8,Z.9,Z.10,Z.11,Z.12,Z.13,Z.14,
        Z.15,Z.16,Z.17,Z.18,Z.19,Z.20,Z.21,Z.22,Z.23,Z.24,Z.25,Z.26,Z.27,
        Z.28,Z.29,Z.30,Z.31;
LEVEL FUNCTION;
DEFINE
BUFFER(SIGN-BIT) = B-BUF-PWR(IMMEDIATE.8);
MUX(Z.0,Z.1,Z.2,Z.3,Z.4,Z.5,Z.6,Z.7,Z.8,Z.9,Z.10,Z.11,Z.12,Z.13,Z.14,Z.15,
   Z.16,Z.17,Z.18,Z.19,Z.20,Z.21,Z.22,Z.23,Z.24,Z.25,Z.26,Z.27,Z.28,Z.29,
   Z.30,Z.31)
  = TV-IF_32(SELECT-IMMEDIATE,IMMEDIATE.0,IMMEDIATE.1,IMMEDIATE.2,
             IMMEDIATE.3,IMMEDIATE.4,IMMEDIATE.5,IMMEDIATE.6,IMMEDIATE.7,
             IMMEDIATE.8,SIGN-BIT,SIGN-BIT,SIGN-BIT,SIGN-BIT,SIGN-BIT,
             SIGN-BIT,SIGN-BIT,SIGN-BIT,SIGN-BIT,SIGN-BIT,SIGN-BIT,SIGN-BIT,
             SIGN-BIT,SIGN-BIT,SIGN-BIT,SIGN-BIT,SIGN-BIT,SIGN-BIT,SIGN-BIT,
             SIGN-BIT,SIGN-BIT,SIGN-BIT,SIGN-BIT,REG-DATA.0,REG-DATA.1,
             REG-DATA.2,REG-DATA.3,REG-DATA.4,REG-DATA.5,REG-DATA.6,
             REG-DATA.7,REG-DATA.8,REG-DATA.9,REG-DATA.10,REG-DATA.11,
             REG-DATA.12,REG-DATA.13,REG-DATA.14,REG-DATA.15,REG-DATA.16,
             REG-DATA.17,REG-DATA.18,REG-DATA.19,REG-DATA.20,REG-DATA.21,
             REG-DATA.22,REG-DATA.23,REG-DATA.24,REG-DATA.25,REG-DATA.26,
             REG-DATA.27,REG-DATA.28,REG-DATA.29,REG-DATA.30,REG-DATA.31);
END MODULE;

MODULE DEC-PASS_32;
INPUTS C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,A.13,A.14,
       A.15,A.16,A.17,A.18,A.19,A.20,A.21,A.22,A.23,A.24,A.25,A.26,A.27,A.28,
       A.29,A.30,A.31;
OUTPUTS Z.0,Z.1,Z.2,Z.3,Z.4,Z.5,Z.6,Z.7,Z.8,Z.9,Z.10,Z.11,Z.12,Z.13,Z.14,
        Z.15,Z.16,Z.17,Z.18,Z.19,Z.20,Z.21,Z.22,Z.23,Z.24,Z.25,Z.26,Z.27,
        Z.28,Z.29,Z.30,Z.31;
LEVEL FUNCTION;
DEFINE
M0(CN) = IVA(C);
M1(Z.0,Z.1,Z.2,Z.3,Z.4,Z.5,Z.6,Z.7,Z.8,Z.9,Z.10,Z.11,Z.12,Z.13,Z.14,Z.15,
   Z.16,Z.17,Z.18,Z.19,Z.20,Z.21,Z.22,Z.23,Z.24,Z.25,Z.26,Z.27,Z.28,Z.29,
   Z.30,Z.31)
  = TV-DEC-PASS-NG_32(CN,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,
                      A.12,A.13,A.14,A.15,A.16,A.17,A.18,A.19,A.20,A.21,
                      A.22,A.23,A.24,A.25,A.26,A.27,A.28,A.29,A.30,A.31);
END MODULE;

MODULE TV-DEC-PASS-NG_32;
INPUTS C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,A.13,A.14,
       A.15,A.16,A.17,A.18,A.19,A.20,A.21,A.22,A.23,A.24,A.25,A.26,A.27,A.28,
       A.29,A.30,A.31;
OUTPUTS Z.0,Z.1,Z.2,Z.3,Z.4,Z.5,Z.6,Z.7,Z.8,Z.9,Z.10,Z.11,Z.12,Z.13,Z.14,
        Z.15,Z.16,Z.17,Z.18,Z.19,Z.20,Z.21,Z.22,Z.23,Z.24,Z.25,Z.26,Z.27,
        Z.28,Z.29,Z.30,Z.31;
LEVEL FUNCTION;
DEFINE
LEFT(GL,Z.0,Z.1,Z.2,Z.3,Z.4,Z.5,Z.6,Z.7,Z.8,Z.9,Z.10,Z.11,Z.12,Z.13,Z.14,
     Z.15)
  = TV-DEC-PASS-G_16(C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,
```

```
                     A.12,A.13,A.14,A.15);
CARRY(CX) = OR2(C,GL);
RIGHT(Z.16,Z.17,Z.18,Z.19,Z.20,Z.21,Z.22,Z.23,Z.24,Z.25,Z.26,Z.27,Z.28,Z.29,
      Z.30,Z.31)
  = TV-DEC-PASS-NG_16(CX,A.16,A.17,A.18,A.19,A.20,A.21,A.22,A.23,A.24,A.25,
                      A.26,A.27,A.28,A.29,A.30,A.31);
END MODULE;

MODULE TV-DEC-PASS-G_16;
INPUTS C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,A.13,A.14,
       A.15;
OUTPUTS G,Z.0,Z.1,Z.2,Z.3,Z.4,Z.5,Z.6,Z.7,Z.8,Z.9,Z.10,Z.11,Z.12,Z.13,Z.14,
        Z.15;
LEVEL FUNCTION;
DEFINE
LEFT(GL,Z.0,Z.1,Z.2,Z.3,Z.4,Z.5,Z.6,Z.7)
  = TV-DEC-PASS-G_8(C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7);
CARRY(CX) = OR2(C,GL);
RIGHT(GR,Z.8,Z.9,Z.10,Z.11,Z.12,Z.13,Z.14,Z.15)
  = TV-DEC-PASS-G_8(CX,A.8,A.9,A.10,A.11,A.12,A.13,A.14,A.15);
GENERATE(G) = OR2(GL,GR);
END MODULE;

MODULE TV-DEC-PASS-NG_16;
INPUTS C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,A.13,A.14,
       A.15;
OUTPUTS Z.0,Z.1,Z.2,Z.3,Z.4,Z.5,Z.6,Z.7,Z.8,Z.9,Z.10,Z.11,Z.12,Z.13,Z.14,
        Z.15;
LEVEL FUNCTION;
DEFINE
LEFT(GL,Z.0,Z.1,Z.2,Z.3,Z.4,Z.5,Z.6,Z.7)
  = TV-DEC-PASS-G_8(C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7);
CARRY(CX) = OR2(C,GL);
RIGHT(Z.8,Z.9,Z.10,Z.11,Z.12,Z.13,Z.14,Z.15)
  = TV-DEC-PASS-NG_8(CX,A.8,A.9,A.10,A.11,A.12,A.13,A.14,A.15);
END MODULE;

MODULE TV-DEC-PASS-G_8;
INPUTS C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7;
OUTPUTS G,Z.0,Z.1,Z.2,Z.3,Z.4,Z.5,Z.6,Z.7;
LEVEL FUNCTION;
DEFINE
LEFT(GL,Z.0,Z.1,Z.2,Z.3) = TV-DEC-PASS-G_4(C,A.0,A.1,A.2,A.3);
CARRY(CX) = OR2(C,GL);
RIGHT(GR,Z.4,Z.5,Z.6,Z.7) = TV-DEC-PASS-G_4(CX,A.4,A.5,A.6,A.7);
GENERATE(G) = OR2(GL,GR);
END MODULE;

MODULE TV-DEC-PASS-NG_8;
INPUTS C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7;
OUTPUTS Z.0,Z.1,Z.2,Z.3,Z.4,Z.5,Z.6,Z.7;
LEVEL FUNCTION;
DEFINE
LEFT(GL,Z.0,Z.1,Z.2,Z.3) = TV-DEC-PASS-G_4(C,A.0,A.1,A.2,A.3);
CARRY(CX) = OR2(C,GL);
```

```
RIGHT(Z.4,Z.5,Z.6,Z.7) = TV-DEC-PASS-NG_4(CX,A.4,A.5,A.6,A.7);
END MODULE;

MODULE TV-DEC-PASS-G_4;
INPUTS C,A.0,A.1,A.2,A.3;
OUTPUTS G,Z.0,Z.1,Z.2,Z.3;
LEVEL FUNCTION;
DEFINE
LEFT(GL,Z.0,Z.1) = TV-DEC-PASS-G_2(C,A.0,A.1);
CARRY(CX) = OR2(C,GL);
RIGHT(GR,Z.2,Z.3) = TV-DEC-PASS-G_2(CX,A.2,A.3);
GENERATE(G) = OR2(GL,GR);
END MODULE;

MODULE TV-DEC-PASS-NG_4;
INPUTS C,A.0,A.1,A.2,A.3;
OUTPUTS Z.0,Z.1,Z.2,Z.3;
LEVEL FUNCTION;
DEFINE
LEFT(GL,Z.0,Z.1) = TV-DEC-PASS-G_2(C,A.0,A.1);
CARRY(CX) = OR2(C,GL);
RIGHT(Z.2,Z.3) = TV-DEC-PASS-NG_2(CX,A.2,A.3);
END MODULE;

MODULE TV-DEC-PASS-G_2;
INPUTS C,A.0,A.1;
OUTPUTS G,Z.0,Z.1;
LEVEL FUNCTION;
DEFINE
LEFT(GL,Z.0) = TV-DEC-PASS-G_1(C,A.0);
CARRY(CX) = OR2(C,GL);
RIGHT(GR,Z.1) = TV-DEC-PASS-G_1(CX,A.1);
GENERATE(G) = OR2(GL,GR);
END MODULE;

MODULE TV-DEC-PASS-NG_2;
INPUTS C,A.0,A.1;
OUTPUTS Z.0,Z.1;
LEVEL FUNCTION;
DEFINE
LEFT(GL,Z.0) = TV-DEC-PASS-G_1(C,A.0);
CARRY(CX) = OR2(C,GL);
RIGHT(Z.1) = TV-DEC-PASS-NG_1(CX,A.1);
END MODULE;

MODULE TV-DEC-PASS-G_1;
INPUTS C,A.0;
OUTPUTS G,Z.0;
LEVEL FUNCTION;
DEFINE
LEAF(G,Z.0) = DEC-PASS-CELL(C,A.0);
END MODULE;

MODULE TV-DEC-PASS-NG_1;
INPUTS C,A.0;
```

```
OUTPUTS Z.0;
LEVEL FUNCTION;
DEFINE
LEAF(G,Z.0) = DEC-PASS-CELL(C,A.0);
END MODULE;

MODULE DEC-PASS-CELL;
INPUTS C,A;
OUTPUTS G,Z;
LEVEL FUNCTION;
DEFINE
G0(G) = ID(A);
G1(Z) = EN(A,C);
END MODULE;

MODULE CORE-ALU_32;
INPUTS C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,A.13,A.14,
       A.15,A.16,A.17,A.18,A.19,A.20,A.21,A.22,A.23,A.24,A.25,A.26,A.27,A.28,
       A.29,A.30,A.31,B.0,B.1,B.2,B.3,B.4,B.5,B.6,B.7,B.8,B.9,B.10,B.11,B.12,
       B.13,B.14,B.15,B.16,B.17,B.18,B.19,B.20,B.21,B.22,B.23,B.24,B.25,B.26,
       B.27,B.28,B.29,B.30,B.31,ZERO,MPG.0,MPG.1,MPG.2,MPG.3,MPG.4,MPG.5,
       MPG.6,OP.0,OP.1,OP.2,OP.3;
OUTPUTS CARRY,OVERFLOW,ZEROP,OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7,
        OUT.8,OUT.9,OUT.10,OUT.11,OUT.12,OUT.13,OUT.14,OUT.15,OUT.16,OUT.17,
        OUT.18,OUT.19,OUT.20,OUT.21,OUT.22,OUT.23,OUT.24,OUT.25,OUT.26,
        OUT.27,OUT.28,OUT.29,OUT.30,OUT.31;
LEVEL FUNCTION;
DEFINE
M-ALU(P,G,ALU-OUT.0,ALU-OUT.1,ALU-OUT.2,ALU-OUT.3,ALU-OUT.4,ALU-OUT.5,
      ALU-OUT.6,ALU-OUT.7,ALU-OUT.8,ALU-OUT.9,ALU-OUT.10,ALU-OUT.11,
      ALU-OUT.12,ALU-OUT.13,ALU-OUT.14,ALU-OUT.15,ALU-OUT.16,ALU-OUT.17,
      ALU-OUT.18,ALU-OUT.19,ALU-OUT.20,ALU-OUT.21,ALU-OUT.22,ALU-OUT.23,
      ALU-OUT.24,ALU-OUT.25,ALU-OUT.26,ALU-OUT.27,ALU-OUT.28,ALU-OUT.29,
      ALU-OUT.30,ALU-OUT.31)
  = TV-ALU-HELP_32(C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,
                   A.13,A.14,A.15,A.16,A.17,A.18,A.19,A.20,A.21,A.22,A.23,
                   A.24,A.25,A.26,A.27,A.28,A.29,A.30,A.31,B.0,B.1,B.2,B.3,
                   B.4,B.5,B.6,B.7,B.8,B.9,B.10,B.11,B.12,B.13,B.14,B.15,
                   B.16,B.17,B.18,B.19,B.20,B.21,B.22,B.23,B.24,B.25,B.26,
                   B.27,B.28,B.29,B.30,B.31,MPG.0,MPG.1,MPG.2,MPG.3,MPG.4,
                   MPG.5,MPG.6);
M-ALU-CARRY(ALU-CARRY) = T-CARRY(C,P,G);
M-CARRY-OUT-HELP(CARRY)
  = CARRY-OUT-HELP(A.0,ALU-CARRY,ZERO,OP.0,OP.1,OP.2,OP.3);
M-OVERFLOW-HELP(OVERFLOW)
  = OVERFLOW-HELP(ALU-OUT.31,A.31,B.31,ZERO,OP.0,OP.1,OP.2,OP.3);
M-SHIFT(OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7,OUT.8,OUT.9,OUT.10,
        OUT.11,OUT.12,OUT.13,OUT.14,OUT.15,OUT.16,OUT.17,OUT.18,OUT.19,
        OUT.20,OUT.21,OUT.22,OUT.23,OUT.24,OUT.25,OUT.26,OUT.27,OUT.28,
        OUT.29,OUT.30,OUT.31)
  = TV-SHIFT-OR-BUF_32(C,ALU-OUT.0,ALU-OUT.1,ALU-OUT.2,ALU-OUT.3,ALU-OUT.4,
                       ALU-OUT.5,ALU-OUT.6,ALU-OUT.7,ALU-OUT.8,ALU-OUT.9,
                       ALU-OUT.10,ALU-OUT.11,ALU-OUT.12,ALU-OUT.13,
                       ALU-OUT.14,ALU-OUT.15,ALU-OUT.16,ALU-OUT.17,
                       ALU-OUT.18,ALU-OUT.19,ALU-OUT.20,ALU-OUT.21,
```

```
                      ALU-OUT.22,ALU-OUT.23,ALU-OUT.24,ALU-OUT.25,
                      ALU-OUT.26,ALU-OUT.27,ALU-OUT.28,ALU-OUT.29,
                      ALU-OUT.30,ALU-OUT.31,A.31,ZERO,OP.0,OP.1,OP.2,OP.3);
M-ZEROP(ZEROP)
  = FAST-ZERO_32(OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7,OUT.8,
                OUT.9,OUT.10,OUT.11,OUT.12,OUT.13,OUT.14,OUT.15,OUT.16,
                OUT.17,OUT.18,OUT.19,OUT.20,OUT.21,OUT.22,OUT.23,OUT.24,
                OUT.25,OUT.26,OUT.27,OUT.28,OUT.29,OUT.30,OUT.31);
END MODULE;

MODULE TV-ALU-HELP_32;
INPUTS C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,A.13,A.14,
       A.15,A.16,A.17,A.18,A.19,A.20,A.21,A.22,A.23,A.24,A.25,A.26,A.27,A.28,
       A.29,A.30,A.31,B.0,B.1,B.2,B.3,B.4,B.5,B.6,B.7,B.8,B.9,B.10,B.11,B.12,
       B.13,B.14,B.15,B.16,B.17,B.18,B.19,B.20,B.21,B.22,B.23,B.24,B.25,B.26,
       B.27,B.28,B.29,B.30,B.31,MPG.0,MPG.1,MPG.2,MPG.3,MPG.4,MPG.5,MPG.6;
OUTPUTS P,G,OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7,OUT.8,OUT.9,
        OUT.10,OUT.11,OUT.12,OUT.13,OUT.14,OUT.15,OUT.16,OUT.17,OUT.18,
        OUT.19,OUT.20,OUT.21,OUT.22,OUT.23,OUT.24,OUT.25,OUT.26,OUT.27,
        OUT.28,OUT.29,OUT.30,OUT.31;
LEVEL FUNCTION;
DEFINE
LHS(PL,GL,OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7,OUT.8,OUT.9,OUT.10,
    OUT.11,OUT.12,OUT.13,OUT.14,OUT.15)
  = TV-ALU-HELP_16(C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,
                  A.13,A.14,A.15,B.0,B.1,B.2,B.3,B.4,B.5,B.6,B.7,B.8,B.9,
                  B.10,B.11,B.12,B.13,B.14,B.15,MPG.0,MPG.1,MPG.2,MPG.3,
                  MPG.4,MPG.5,MPG.6);
LHS-CARRY(CL) = T-CARRY(C,PL,GL);
RHS(PR,GR,OUT.16,OUT.17,OUT.18,OUT.19,OUT.20,OUT.21,OUT.22,OUT.23,OUT.24,
    OUT.25,OUT.26,OUT.27,OUT.28,OUT.29,OUT.30,OUT.31)
  = TV-ALU-HELP_16(CL,A.16,A.17,A.18,A.19,A.20,A.21,A.22,A.23,A.24,A.25,
                  A.26,A.27,A.28,A.29,A.30,A.31,B.16,B.17,B.18,B.19,B.20,
                  B.21,B.22,B.23,B.24,B.25,B.26,B.27,B.28,B.29,B.30,B.31,
                  MPG.0,MPG.1,MPG.2,MPG.3,MPG.4,MPG.5,MPG.6);
P(P) = AN2(PL,PR);
G(G) = T-CARRY(GL,PR,GR);
END MODULE;

MODULE TV-ALU-HELP_16;
INPUTS C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,A.13,A.14,
       A.15,B.0,B.1,B.2,B.3,B.4,B.5,B.6,B.7,B.8,B.9,B.10,B.11,B.12,B.13,B.14,
       B.15,MPG.0,MPG.1,MPG.2,MPG.3,MPG.4,MPG.5,MPG.6;
OUTPUTS P,G,OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7,OUT.8,OUT.9,
        OUT.10,OUT.11,OUT.12,OUT.13,OUT.14,OUT.15;
LEVEL FUNCTION;
DEFINE
LHS(PL,GL,OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7)
  = TV-ALU-HELP_8(C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,B.0,B.1,B.2,B.3,B.4,B.5,
                 B.6,B.7,MPG.0,MPG.1,MPG.2,MPG.3,MPG.4,MPG.5,MPG.6);
LHS-CARRY(CL) = T-CARRY(C,PL,GL);
RHS(PR,GR,OUT.8,OUT.9,OUT.10,OUT.11,OUT.12,OUT.13,OUT.14,OUT.15)
  = TV-ALU-HELP_8(CL,A.8,A.9,A.10,A.11,A.12,A.13,A.14,A.15,B.8,B.9,B.10,
                 B.11,B.12,B.13,B.14,B.15,MPG.0,MPG.1,MPG.2,MPG.3,MPG.4,
                 MPG.5,MPG.6);
```

```
P(P) = AN2(PL,PR);
G(G) = T-CARRY(GL,PR,GR);
END MODULE;

MODULE TV-ALU-HELP_8;
INPUTS C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,B.0,B.1,B.2,B.3,B.4,B.5,B.6,B.7,
       MPG.0,MPG.1,MPG.2,MPG.3,MPG.4,MPG.5,MPG.6;
OUTPUTS P,G,OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7;
LEVEL FUNCTION;
DEFINE
BUFFERMPG(MPG-.0,MPG-.1,MPG-.2,MPG-.3,MPG-.4,MPG-.5,MPG-.6)
  = V-BUF_7(MPG.0,MPG.1,MPG.2,MPG.3,MPG.4,MPG.5,MPG.6);
LHS(PL,GL,OUT.0,OUT.1,OUT.2,OUT.3)
  = TV-ALU-HELP_4(C,A.0,A.1,A.2,A.3,B.0,B.1,B.2,B.3,MPG-.0,MPG-.1,MPG-.2,
                  MPG-.3,MPG-.4,MPG-.5,MPG-.6);
LHS-CARRY(CL) = T-CARRY(C,PL,GL);
RHS(PR,GR,OUT.4,OUT.5,OUT.6,OUT.7)
  = TV-ALU-HELP_4(CL,A.4,A.5,A.6,A.7,B.4,B.5,B.6,B.7,MPG-.0,MPG-.1,MPG-.2,
                  MPG-.3,MPG-.4,MPG-.5,MPG-.6);
P(P) = AN2(PL,PR);
G(G) = T-CARRY(GL,PR,GR);
END MODULE;

MODULE TV-ALU-HELP_4;
INPUTS C,A.0,A.1,A.2,A.3,B.0,B.1,B.2,B.3,MPG.0,MPG.1,MPG.2,MPG.3,MPG.4,MPG.5,
       MPG.6;
OUTPUTS P,G,OUT.0,OUT.1,OUT.2,OUT.3;
LEVEL FUNCTION;
DEFINE
LHS(PL,GL,OUT.0,OUT.1)
  = TV-ALU-HELP_2(C,A.0,A.1,B.0,B.1,MPG.0,MPG.1,MPG.2,MPG.3,MPG.4,MPG.5,
                  MPG.6);
LHS-CARRY(CL) = T-CARRY(C,PL,GL);
RHS(PR,GR,OUT.2,OUT.3)
  = TV-ALU-HELP_2(CL,A.2,A.3,B.2,B.3,MPG.0,MPG.1,MPG.2,MPG.3,MPG.4,MPG.5,
                  MPG.6);
P(P) = AN2(PL,PR);
G(G) = T-CARRY(GL,PR,GR);
END MODULE;

MODULE TV-ALU-HELP_2;
INPUTS C,A.0,A.1,B.0,B.1,MPG.0,MPG.1,MPG.2,MPG.3,MPG.4,MPG.5,MPG.6;
OUTPUTS P,G,OUT.0,OUT.1;
LEVEL FUNCTION;
DEFINE
LHS(PL,GL,OUT.0)
  = TV-ALU-HELP_1(C,A.0,B.0,MPG.0,MPG.1,MPG.2,MPG.3,MPG.4,MPG.5,MPG.6);
LHS-CARRY(CL) = T-CARRY(C,PL,GL);
RHS(PR,GR,OUT.1)
  = TV-ALU-HELP_1(CL,A.1,B.1,MPG.0,MPG.1,MPG.2,MPG.3,MPG.4,MPG.5,MPG.6);
P(P) = AN2(PL,PR);
G(G) = T-CARRY(GL,PR,GR);
END MODULE;

MODULE TV-ALU-HELP_1;
```

```
INPUTS C,A.0,B.0,MPG.0,MPG.1,MPG.2,MPG.3,MPG.4,MPG.5,MPG.6;
OUTPUTS P,G,OUT.0;
LEVEL FUNCTION;
DEFINE
LEAF(P,G,OUT.0)
  = ALU-CELL(C,A.0,B.0,MPG.0,MPG.1,MPG.2,MPG.3,MPG.4,MPG.5,MPG.6);
END MODULE;

MODULE ALU-CELL;
INPUTS C,A,B,GBN,GAN,GA,PB,PAN,PA,M;
OUTPUTS P,G,Z;
LEVEL FUNCTION;
DEFINE
N0(AN) = IVA(A);
N1(BN) = IVA(B);
P0(P) = P-CELL(A,AN,B,PA,PAN,PB);
G0(G) = G-CELL(A,AN,BN,GA,GAN,GBN);
M0(MC) = ND2(C,M);
Z0(Z) = EN3(MC,P,G);
END MODULE;

MODULE P-CELL;
INPUTS A,AN,B,PA,PAN,PB;
OUTPUTS W-3;
LEVEL FUNCTION;
DEFINE
G-2(W-2) = ND2(B,PB);
G-1(W-1) = ND2(AN,PAN);
G-0(W-0) = ND2(A,PA);
G-3(W-3) = ND3(W-0,W-1,W-2);
END MODULE;

MODULE G-CELL;
INPUTS A,AN,BN,GA,GAN,GBN;
OUTPUTS W-3;
LEVEL FUNCTION;
DEFINE
G-2(W-2) = ND2(BN,GBN);
G-1(W-1) = ND2(AN,GAN);
G-0(W-0) = ND2(A,GA);
G-3(W-3) = AN3(W-0,W-1,W-2);
END MODULE;

MODULE V-BUF_7;
INPUTS A.0,A.1,A.2,A.3,A.4,A.5,A.6;
OUTPUTS Y.0,Y.1,Y.2,Y.3,Y.4,Y.5,Y.6;
LEVEL FUNCTION;
DEFINE
G.0(Y.0) = B-BUF(A.0);
G.1(Y.1) = B-BUF(A.1);
G.2(Y.2) = B-BUF(A.2);
G.3(Y.3) = B-BUF(A.3);
G.4(Y.4) = B-BUF(A.4);
G.5(Y.5) = B-BUF(A.5);
G.6(Y.6) = B-BUF(A.6);
```

```
END MODULE;

MODULE T-CARRY;
INPUTS C,PROP,GEN;
OUTPUTS Z;
LEVEL FUNCTION;
DEFINE
G0(Z-) = AO6(C,PROP,GEN);
G1(Z) = IVA(Z-);
END MODULE;

MODULE CARRY-OUT-HELP;
INPUTS A0,RESULT,ZERO,OP0,OP1,OP2,OP3;
OUTPUTS W-22;
LEVEL FUNCTION;
DEFINE
G-21(W-21) = IVA(ZERO);
G-16(W-16) = IVA(OP1);
G-17(W-17) = IVA(W-16);
G-14(W-14) = IVA(OP0);
G-15(W-15) = IVA(W-14);
G-18(W-18) = ND2(W-15,W-17);
G-13(W-13) = IVA(OP2);
G-11(W-11) = IVA(OP3);
G-12(W-12) = IVA(W-11);
G-19(W-19) = ND4(W-12,W-13,W-18,A0);
G-9(W-9) = IVA(RESULT);
G-8(W-8) = IVA(W-13);
G-10(W-10) = ND3(W-11,W-8,W-9);
G-3(W-3) = ND2(W-14,W-16);
G-5(W-5) = ND4(W-11,W-3,W-13,RESULT);
G-20(W-20) = ND3(W-5,W-10,W-19);
G-22(W-22) = AN2(W-20,W-21);
END MODULE;

MODULE OVERFLOW-HELP;
INPUTS RN,AN,BN,ZERO,OP0,OP1,OP2,OP3;
OUTPUTS W-83;
LEVEL FUNCTION;
DEFINE
G-77(W-77) = IVA(AN);
G-78(W-78) = IVA(W-77);
G-76(W-76) = IVA(OP2);
G-79(W-79) = ND2(W-76,W-78);
G-72(W-72) = IVA(OP1);
G-75(W-75) = ND3(OP0,W-72,W-78);
G-69(W-69) = IVA(W-76);
G-71(W-71) = ND2(W-69,W-77);
G-80(W-80) = ND3(W-71,W-75,W-79);
G-81(W-81) = IVA(W-80);
G-66(W-66) = IVA(ZERO);
G-63(W-63) = ND3(W-72,W-76,W-78);
G-57(W-57) = EO(W-78,BN);
G-58(W-58) = ND3(OP1,W-76,W-57);
G-64(W-64) = ND2(W-58,W-63);
```

```
G-52(W-52) = ND3(W-72,W-69,W-77);
G-47(W-47) = OR3(W-72,W-76,W-57);
G-41(W-41) = IVA(OP3);
G-53(W-53) = ND3(W-41,W-47,W-52);
G-65(W-65) = NR2(W-53,W-64);
G-67(W-67) = ND2(W-65,W-66);
G-82(W-82) = NR2(W-67,W-81);
G-40(W-40) = NR2(W-67,W-80);
G-83(W-83) = MUX21H(W-82,W-40,RN);
END MODULE;

MODULE TV-SHIFT-OR-BUF_32;
INPUTS C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,A.13,A.14,
       A.15,A.16,A.17,A.18,A.19,A.20,A.21,A.22,A.23,A.24,A.25,A.26,A.27,A.28,
       A.29,A.30,A.31,AN,ZERO,OP0,OP1,OP2,OP3;
OUTPUTS OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7,OUT.8,OUT.9,OUT.10,
        OUT.11,OUT.12,OUT.13,OUT.14,OUT.15,OUT.16,OUT.17,OUT.18,OUT.19,
        OUT.20,OUT.21,OUT.22,OUT.23,OUT.24,OUT.25,OUT.26,OUT.27,OUT.28,
        OUT.29,OUT.30,OUT.31;
LEVEL FUNCTION;
DEFINE
CNTL(CTL,SI) = SHIFT-OR-BUF-CNTL(C,AN,ZERO,OP0,OP1,OP2,OP3);
MUX(OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7,OUT.8,OUT.9,OUT.10,
    OUT.11,OUT.12,OUT.13,OUT.14,OUT.15,OUT.16,OUT.17,OUT.18,OUT.19,OUT.20,
    OUT.21,OUT.22,OUT.23,OUT.24,OUT.25,OUT.26,OUT.27,OUT.28,OUT.29,OUT.30,
    OUT.31)
  = TV-IF_32(CTL,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,
             A.13,A.14,A.15,A.16,A.17,A.18,A.19,A.20,A.21,A.22,A.23,A.24,
             A.25,A.26,A.27,A.28,A.29,A.30,A.31,A.1,A.2,A.3,A.4,A.5,A.6,A.7,
             A.8,A.9,A.10,A.11,A.12,A.13,A.14,A.15,A.16,A.17,A.18,A.19,A.20,
             A.21,A.22,A.23,A.24,A.25,A.26,A.27,A.28,A.29,A.30,A.31,SI);
END MODULE;

MODULE TV-IF_32;
INPUTS C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,A.13,A.14,
       A.15,A.16,A.17,A.18,A.19,A.20,A.21,A.22,A.23,A.24,A.25,A.26,A.27,A.28,
       A.29,A.30,A.31,B.0,B.1,B.2,B.3,B.4,B.5,B.6,B.7,B.8,B.9,B.10,B.11,B.12,
       B.13,B.14,B.15,B.16,B.17,B.18,B.19,B.20,B.21,B.22,B.23,B.24,B.25,B.26,
       B.27,B.28,B.29,B.30,B.31;
OUTPUTS OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7,OUT.8,OUT.9,OUT.10,
        OUT.11,OUT.12,OUT.13,OUT.14,OUT.15,OUT.16,OUT.17,OUT.18,OUT.19,
        OUT.20,OUT.21,OUT.22,OUT.23,OUT.24,OUT.25,OUT.26,OUT.27,OUT.28,
        OUT.29,OUT.30,OUT.31;
LEVEL FUNCTION;
DEFINE
C-BUF(C-BUF) = B-BUF(C);
LEFT(OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7,OUT.8,OUT.9,OUT.10,
     OUT.11,OUT.12,OUT.13,OUT.14,OUT.15)
  = TV-IF_16(C-BUF,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,
             A.13,A.14,A.15,B.0,B.1,B.2,B.3,B.4,B.5,B.6,B.7,B.8,B.9,B.10,
             B.11,B.12,B.13,B.14,B.15);
RIGHT(OUT.16,OUT.17,OUT.18,OUT.19,OUT.20,OUT.21,OUT.22,OUT.23,OUT.24,OUT.25,
      OUT.26,OUT.27,OUT.28,OUT.29,OUT.30,OUT.31)
  = TV-IF_16(C-BUF,A.16,A.17,A.18,A.19,A.20,A.21,A.22,A.23,A.24,A.25,A.26,
             A.27,A.28,A.29,A.30,A.31,B.16,B.17,B.18,B.19,B.20,B.21,B.22,
```

```
            B.23,B.24,B.25,B.26,B.27,B.28,B.29,B.30,B.31);
END MODULE;

MODULE TV-IF_16;
INPUTS C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,A.13,A.14,
       A.15,B.0,B.1,B.2,B.3,B.4,B.5,B.6,B.7,B.8,B.9,B.10,B.11,B.12,B.13,B.14,
       B.15;
OUTPUTS OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7,OUT.8,OUT.9,OUT.10,
        OUT.11,OUT.12,OUT.13,OUT.14,OUT.15;
LEVEL FUNCTION;
DEFINE
LEFT(OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7)
  = TV-IF_8(C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,B.0,B.1,B.2,B.3,B.4,B.5,B.6,
           B.7);
RIGHT(OUT.8,OUT.9,OUT.10,OUT.11,OUT.12,OUT.13,OUT.14,OUT.15)
  = TV-IF_8(C,A.8,A.9,A.10,A.11,A.12,A.13,A.14,A.15,B.8,B.9,B.10,B.11,B.12,
           B.13,B.14,B.15);
END MODULE;

MODULE TV-IF_8;
INPUTS C,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,B.0,B.1,B.2,B.3,B.4,B.5,B.6,B.7;
OUTPUTS OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7;
LEVEL FUNCTION;
DEFINE
LEFT(OUT.0,OUT.1,OUT.2,OUT.3) = TV-IF_4(C,A.0,A.1,A.2,A.3,B.0,B.1,B.2,B.3);
RIGHT(OUT.4,OUT.5,OUT.6,OUT.7) = TV-IF_4(C,A.4,A.5,A.6,A.7,B.4,B.5,B.6,B.7);
END MODULE;

MODULE SHIFT-OR-BUF-CNTL;
INPUTS C,AN,ZERO,OP0,OP1,OP2,OP3;
OUTPUTS W-3,W-9;
LEVEL FUNCTION;
DEFINE
G-6(W-6) = IVA(OP1);
G-5(W-5) = IVA(OP0);
G-7(W-7) = AN2(W-5,W-6);
G-8(W-8) = AN2(W-7,C);
G-4(W-4) = AN2(OP0,AN);
G-9(W-9) = OR2(W-4,W-8);
G-1(W-1) = IVA(OP2);
G-2(W-2) = ND2(W-1,OP3);
G-0(W-0) = AN2(OP0,OP1);
G-3(W-3) = OR3(W-0,W-2,ZERO);
END MODULE;

MODULE FAST-ZERO_32;
INPUTS A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,A.13,A.14,A.15,
       A.16,A.17,A.18,A.19,A.20,A.21,A.22,A.23,A.24,A.25,A.26,A.27,A.28,A.29,
       A.30,A.31;
OUTPUTS Z;
LEVEL FUNCTION;
DEFINE
FRONT(ZFRONT)
  = T-OR_34358165488(A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,
                    A.13,A.14,A.15,A.16,A.17,A.18,A.19,A.20,A.21,A.22,A.23,
```

```
                     A.24,A.25,A.26,A.27,A.28,A.29);
RESULT(Z) = NR3(ZFRONT,A.30,A.31);
END MODULE;

MODULE T-OR_34358165488;
INPUTS A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,A.13,A.14,A.15,
       A.16,A.17,A.18,A.19,A.20,A.21,A.22,A.23,A.24,A.25,A.26,A.27,A.28,A.29;
OUTPUTS OUT;
LEVEL FUNCTION;
DEFINE
LEFT(LEFT-OUT)
  = T-NOR_262136(A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,
                 A.13,A.14);
RIGHT(RIGHT-OUT)
  = T-NOR_262136(A.15,A.16,A.17,A.18,A.19,A.20,A.21,A.22,A.23,A.24,A.25,
                 A.26,A.27,A.28,A.29);
OUTPUT(OUT) = ND2(LEFT-OUT,RIGHT-OUT);
END MODULE;

MODULE T-NOR_262136;
INPUTS A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7,A.8,A.9,A.10,A.11,A.12,A.13,A.14;
OUTPUTS OUT;
LEVEL FUNCTION;
DEFINE
LEFT(LEFT-OUT) = T-OR_508(A.0,A.1,A.2,A.3,A.4,A.5,A.6);
RIGHT(RIGHT-OUT) = T-OR_8(A.7,A.8,A.9,A.10,A.11,A.12,A.13,A.14);
OUTPUT(OUT) = NR2(LEFT-OUT,RIGHT-OUT);
END MODULE;

MODULE T-OR_508;
INPUTS A.0,A.1,A.2,A.3,A.4,A.5,A.6;
OUTPUTS OUT;
LEVEL FUNCTION;
DEFINE
LEFT(LEFT-OUT) = T-NOR_14(A.0,A.1,A.2);
RIGHT(RIGHT-OUT) = T-NOR_4(A.3,A.4,A.5,A.6);
OUTPUT(OUT) = ND2(LEFT-OUT,RIGHT-OUT);
END MODULE;

MODULE T-NOR_14;
INPUTS A.0,A.1,A.2;
OUTPUTS OUT;
LEVEL FUNCTION;
DEFINE
LEFT(LEFT-OUT) = T-OR_1(A.0);
RIGHT(RIGHT-OUT) = T-OR_2(A.1,A.2);
OUTPUT(OUT) = NR2(LEFT-OUT,RIGHT-OUT);
END MODULE;

MODULE T-OR_1;
INPUTS A.0;
OUTPUTS OUT;
LEVEL FUNCTION;
DEFINE
LEAF(OUT) = B-BUF(A.0);
```

```
END MODULE;

MODULE T-OR_8;
INPUTS A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7;
OUTPUTS OUT;
LEVEL FUNCTION;
DEFINE
LEFT(LEFT-OUT) = T-NOR_4(A.0,A.1,A.2,A.3);
RIGHT(RIGHT-OUT) = T-NOR_4(A.4,A.5,A.6,A.7);
OUTPUT(OUT) = ND2(LEFT-OUT,RIGHT-OUT);
END MODULE;

MODULE T-NOR_4;
INPUTS A.0,A.1,A.2,A.3;
OUTPUTS OUT;
LEVEL FUNCTION;
DEFINE
LEFT(LEFT-OUT) = T-OR_2(A.0,A.1);
RIGHT(RIGHT-OUT) = T-OR_2(A.2,A.3);
OUTPUT(OUT) = NR2(LEFT-OUT,RIGHT-OUT);
END MODULE;

MODULE T-OR_2;
INPUTS A.0,A.1;
OUTPUTS OUT;
LEVEL FUNCTION;
DEFINE
LEAF(OUT) = OR2(A.0,A.1);
END MODULE;

MODULE NEXT-CNTL-STATE;
INPUTS RESET-,DTACK-,HOLD-,RW-,STATE.0,STATE.1,STATE.2,STATE.3,STATE.4,
       I-REG.0,I-REG.1,I-REG.2,I-REG.3,I-REG.4,I-REG.5,I-REG.6,I-REG.7,
       I-REG.8,I-REG.9,I-REG.10,I-REG.11,I-REG.12,I-REG.13,I-REG.14,I-REG.15,
       I-REG.16,I-REG.17,I-REG.18,I-REG.19,I-REG.20,I-REG.21,I-REG.22,
       I-REG.23,I-REG.24,I-REG.25,I-REG.26,I-REG.27,I-REG.28,I-REG.29,
       I-REG.30,I-REG.31,FLAGS.0,FLAGS.1,FLAGS.2,FLAGS.3,PC-REG.0,PC-REG.1,
       PC-REG.2,PC-REG.3,REGS-ADDRESS.0,REGS-ADDRESS.1,REGS-ADDRESS.2,
       REGS-ADDRESS.3;
OUTPUTS NEXT-CNTL-STATE.0,NEXT-CNTL-STATE.1,NEXT-CNTL-STATE.2,
        NEXT-CNTL-STATE.3,NEXT-CNTL-STATE.4,NEXT-CNTL-STATE.5,
        NEXT-CNTL-STATE.6,NEXT-CNTL-STATE.7,NEXT-CNTL-STATE.8,
        NEXT-CNTL-STATE.9,NEXT-CNTL-STATE.10,NEXT-CNTL-STATE.11,
        NEXT-CNTL-STATE.12,NEXT-CNTL-STATE.13,NEXT-CNTL-STATE.14,
        NEXT-CNTL-STATE.15,NEXT-CNTL-STATE.16,NEXT-CNTL-STATE.17,
        NEXT-CNTL-STATE.18,NEXT-CNTL-STATE.19,NEXT-CNTL-STATE.20,
        NEXT-CNTL-STATE.21,NEXT-CNTL-STATE.22,NEXT-CNTL-STATE.23,
        NEXT-CNTL-STATE.24,NEXT-CNTL-STATE.25,NEXT-CNTL-STATE.26,
        NEXT-CNTL-STATE.27,NEXT-CNTL-STATE.28,NEXT-CNTL-STATE.29,
        NEXT-CNTL-STATE.30,NEXT-CNTL-STATE.31,NEXT-CNTL-STATE.32,
        NEXT-CNTL-STATE.33,NEXT-CNTL-STATE.34,NEXT-CNTL-STATE.35,
        NEXT-CNTL-STATE.36,NEXT-CNTL-STATE.37,NEXT-CNTL-STATE.38,
        NEXT-CNTL-STATE.39;
LEVEL FUNCTION;
DEFINE
```

```
CONTROL-SIGNALS(A-IMMEDIATE-P,STORE,SET-SOME-FLAGS,DIRECT-A,DIRECT-B,UNARY,
                PRE-DEC-A,PRE-DEC-B,C,ALL-T-REGS-ADDRESS,SIDE-EFFECT-A,
                SIDE-EFFECT-B)
  = CONTROL-LET(I-REG.0,I-REG.1,I-REG.2,I-REG.3,I-REG.4,I-REG.5,I-REG.6,
                I-REG.7,I-REG.8,I-REG.9,I-REG.10,I-REG.11,I-REG.12,I-REG.13,
                I-REG.14,I-REG.15,I-REG.16,I-REG.17,I-REG.18,I-REG.19,
                I-REG.20,I-REG.21,I-REG.22,I-REG.23,I-REG.24,I-REG.25,
                I-REG.26,I-REG.27,I-REG.28,I-REG.29,I-REG.30,I-REG.31,
                FLAGS.0,FLAGS.1,FLAGS.2,FLAGS.3,REGS-ADDRESS.0,
                REGS-ADDRESS.1,REGS-ADDRESS.2,REGS-ADDRESS.3);
NOT-RESET(RESET) = IVA(RESET-);
RESET5X(RESET5X.0,RESET5X.1,RESET5X.2,RESET5X.3,RESET5X.4) = FANOUT-5(RESET);
XSTATE(XSTATE.0,XSTATE.1,XSTATE.2,XSTATE.3,XSTATE.4)
  = V-OR_5(RESET5X.0,RESET5X.1,RESET5X.2,RESET5X.3,RESET5X.4,STATE.0,
           STATE.1,STATE.2,STATE.3,STATE.4);
DSTATE(DECODED-STATE.0,DECODED-STATE.1,DECODED-STATE.2,DECODED-STATE.3,
       DECODED-STATE.4,DECODED-STATE.5,DECODED-STATE.6,DECODED-STATE.7,
       DECODED-STATE.8,DECODED-STATE.9,DECODED-STATE.10,DECODED-STATE.11,
       DECODED-STATE.12,DECODED-STATE.13,DECODED-STATE.14,DECODED-STATE.15,
       DECODED-STATE.16,DECODED-STATE.17,DECODED-STATE.18,DECODED-STATE.19,
       DECODED-STATE.20,DECODED-STATE.21,DECODED-STATE.22,DECODED-STATE.23,
       DECODED-STATE.24,DECODED-STATE.25,DECODED-STATE.26,DECODED-STATE.27,
       DECODED-STATE.28,DECODED-STATE.29,DECODED-STATE.30,DECODED-STATE.31)
  = DECODE-5(XSTATE.0,XSTATE.1,XSTATE.2,XSTATE.3,XSTATE.4);
NXSTATE(NEXT-STATE.0,NEXT-STATE.1,NEXT-STATE.2,NEXT-STATE.3,NEXT-STATE.4,
        NEXT-STATE.5,NEXT-STATE.6,NEXT-STATE.7,NEXT-STATE.8,NEXT-STATE.9,
        NEXT-STATE.10,NEXT-STATE.11,NEXT-STATE.12,NEXT-STATE.13,
        NEXT-STATE.14,NEXT-STATE.15,NEXT-STATE.16,NEXT-STATE.17,
        NEXT-STATE.18,NEXT-STATE.19,NEXT-STATE.20,NEXT-STATE.21,
        NEXT-STATE.22,NEXT-STATE.23,NEXT-STATE.24,NEXT-STATE.25,
        NEXT-STATE.26,NEXT-STATE.27,NEXT-STATE.28,NEXT-STATE.29,
        NEXT-STATE.30,NEXT-STATE.31)
  = NEXT-STATE(DECODED-STATE.0,DECODED-STATE.1,DECODED-STATE.2,
               DECODED-STATE.3,DECODED-STATE.4,DECODED-STATE.5,
               DECODED-STATE.6,DECODED-STATE.7,DECODED-STATE.8,
               DECODED-STATE.9,DECODED-STATE.10,DECODED-STATE.11,
               DECODED-STATE.12,DECODED-STATE.13,DECODED-STATE.14,
               DECODED-STATE.15,DECODED-STATE.16,DECODED-STATE.17,
               DECODED-STATE.18,DECODED-STATE.19,DECODED-STATE.20,
               DECODED-STATE.21,DECODED-STATE.22,DECODED-STATE.23,
               DECODED-STATE.24,DECODED-STATE.25,DECODED-STATE.26,
               DECODED-STATE.27,DECODED-STATE.28,DECODED-STATE.29,
               DECODED-STATE.30,DECODED-STATE.31,STORE,SET-SOME-FLAGS,UNARY,
               DIRECT-A,DIRECT-B,SIDE-EFFECT-A,SIDE-EFFECT-B,
               ALL-T-REGS-ADDRESS,DTACK-,HOLD-);
CVECTOR(NEXT-CNTL-STATE.0,NEXT-CNTL-STATE.1,NEXT-CNTL-STATE.2,
        NEXT-CNTL-STATE.3,NEXT-CNTL-STATE.4,NEXT-CNTL-STATE.5,
        NEXT-CNTL-STATE.6,NEXT-CNTL-STATE.7,NEXT-CNTL-STATE.8,
        NEXT-CNTL-STATE.9,NEXT-CNTL-STATE.10,NEXT-CNTL-STATE.11,
        NEXT-CNTL-STATE.12,NEXT-CNTL-STATE.13,NEXT-CNTL-STATE.14,
        NEXT-CNTL-STATE.15,NEXT-CNTL-STATE.16,NEXT-CNTL-STATE.17,
        NEXT-CNTL-STATE.18,NEXT-CNTL-STATE.19,NEXT-CNTL-STATE.20,
        NEXT-CNTL-STATE.21,NEXT-CNTL-STATE.22,NEXT-CNTL-STATE.23,
        NEXT-CNTL-STATE.24,NEXT-CNTL-STATE.25,NEXT-CNTL-STATE.26,
        NEXT-CNTL-STATE.27,NEXT-CNTL-STATE.28,NEXT-CNTL-STATE.29,
```

```
        NEXT-CNTL-STATE.30,NEXT-CNTL-STATE.31,NEXT-CNTL-STATE.32,
        NEXT-CNTL-STATE.33,NEXT-CNTL-STATE.34,NEXT-CNTL-STATE.35,
        NEXT-CNTL-STATE.36,NEXT-CNTL-STATE.37,NEXT-CNTL-STATE.38,
        NEXT-CNTL-STATE.39)
  = CV(NEXT-STATE.0,NEXT-STATE.1,NEXT-STATE.2,NEXT-STATE.3,NEXT-STATE.4,
       NEXT-STATE.5,NEXT-STATE.6,NEXT-STATE.7,NEXT-STATE.8,NEXT-STATE.9,
       NEXT-STATE.10,NEXT-STATE.11,NEXT-STATE.12,NEXT-STATE.13,
       NEXT-STATE.14,NEXT-STATE.15,NEXT-STATE.16,NEXT-STATE.17,
       NEXT-STATE.18,NEXT-STATE.19,NEXT-STATE.20,NEXT-STATE.21,
       NEXT-STATE.22,NEXT-STATE.23,NEXT-STATE.24,NEXT-STATE.25,
       NEXT-STATE.26,NEXT-STATE.27,NEXT-STATE.28,NEXT-STATE.29,
       NEXT-STATE.30,NEXT-STATE.31,I-REG.0,I-REG.1,I-REG.2,I-REG.3,I-REG.10,
       I-REG.11,I-REG.12,I-REG.13,I-REG.24,I-REG.25,I-REG.26,I-REG.27,
       PC-REG.0,PC-REG.1,PC-REG.2,PC-REG.3,REGS-ADDRESS.0,REGS-ADDRESS.1,
       REGS-ADDRESS.2,REGS-ADDRESS.3,I-REG.16,I-REG.17,I-REG.18,I-REG.19,
       STORE,C,A-IMMEDIATE-P,RW-,DIRECT-A,SIDE-EFFECT-A,SIDE-EFFECT-B,
       PRE-DEC-A,PRE-DEC-B);
END MODULE;

MODULE CONTROL-LET;
INPUTS I-REG.0,I-REG.1,I-REG.2,I-REG.3,I-REG.4,I-REG.5,I-REG.6,I-REG.7,
       I-REG.8,I-REG.9,I-REG.10,I-REG.11,I-REG.12,I-REG.13,I-REG.14,I-REG.15,
       I-REG.16,I-REG.17,I-REG.18,I-REG.19,I-REG.20,I-REG.21,I-REG.22,
       I-REG.23,I-REG.24,I-REG.25,I-REG.26,I-REG.27,I-REG.28,I-REG.29,
       I-REG.30,I-REG.31,FLAGS.0,FLAGS.1,FLAGS.2,FLAGS.3,REGS-ADDRESS.0,
       REGS-ADDRESS.1,REGS-ADDRESS.2,REGS-ADDRESS.3;
OUTPUTS A-IMMEDIATE-P,STORE,SET-SOME-FLAGS,DIRECT-A,DIRECT-B,UNARY,PRE-DEC-A,
        PRE-DEC-B,C,ALL-T-REGS-ADDRESS,SIDE-EFFECT-A,SIDE-EFFECT-B;
LEVEL FUNCTION;
DEFINE
G0(A-IMMEDIATE-P) = ID(I-REG.9);
G1(A-IMMEDIATE-P-) = IVA(A-IMMEDIATE-P);
G2(STORE)
  = B-STORE-RESULTP(I-REG.20,I-REG.21,I-REG.22,I-REG.23,FLAGS.0,FLAGS.1,
                    FLAGS.2,FLAGS.3);
G3(SET-SOME-FLAGS) = OR4(I-REG.16,I-REG.17,I-REG.18,I-REG.19);
G4(ALMOST-DIRECT-A,PRE-DEC-A,ALMOST-SIDE-EFFECT-A)
  = DECODE-REG-MODE(I-REG.4,I-REG.5);
G5(DIRECT-B,PRE-DEC-B,SIDE-EFFECT-B) = DECODE-REG-MODE(I-REG.14,I-REG.15);
G6(UNARY) = UNARY-OP-CODE-P(I-REG.24,I-REG.25,I-REG.26,I-REG.27);
G7(C) = ID(FLAGS.3);
G8(ALL-T-REGS-ADDRESS)
  = AN4(REGS-ADDRESS.0,REGS-ADDRESS.1,REGS-ADDRESS.2,REGS-ADDRESS.3);
G9(SIDE-EFFECT-A) = AN2(A-IMMEDIATE-P-,ALMOST-SIDE-EFFECT-A);
GA(DIRECT-A) = OR2(A-IMMEDIATE-P,ALMOST-DIRECT-A);
END MODULE;

MODULE B-STORE-RESULTP;
INPUTS S0,S1,S2,S3,Z,N,V,C;
OUTPUTS RESULT;
LEVEL FUNCTION;
DEFINE
G0(CZ) = OR2(C,Z);
G1(NV) = EO(N,V);
G2(ZNV) = OR2(Z,NV);
```

```
G3(MUX) = STORE-RESULTP-MUX(S1,S2,S3,C,V,N,Z,CZ,NV,ZNV);
G4(RESULT) = EO(S0,MUX);
END MODULE;

MODULE STORE-RESULTP-MUX;
INPUTS S0,S1,S2,D0,D1,D2,D3,D4,D5,D6;
OUTPUTS W-13;
LEVEL FUNCTION;
DEFINE
G-10(W-10) = IVA(S0);
G-11(W-11) = ND2(W-10,D6);
G-9(W-9) = AO2(W-10,D4,S0,D5);
G-7(W-7) = IVA(S1);
G-12(W-12) = AO2(W-7,W-9,S1,W-11);
G-5(W-5) = AO2(W-10,D2,S0,D3);
G-3(W-3) = AO2(W-10,D0,S0,D1);
G-6(W-6) = AO2(W-7,W-3,S1,W-5);
G-0(W-0) = IVA(S2);
G-13(W-13) = AO2(W-0,W-6,S2,W-12);
END MODULE;

MODULE DECODE-REG-MODE;
INPUTS M0,M1;
OUTPUTS DIRECT,PRE-DEC,SIDE-EFFECT;
LEVEL FUNCTION;
DEFINE
G0(DIRECT) = NR2(M0,M1);
G1(M1-) = IVA(M1);
G2(PRE-DEC) = NR2(M0,M1-);
G3(SIDE-EFFECT) = ID(M1);
END MODULE;

MODULE UNARY-OP-CODE-P;
INPUTS OP0,OP1,OP2,OP3;
OUTPUTS Z;
LEVEL FUNCTION;
DEFINE
G0(OP0-) = IVA(OP0);
G1(OP1-) = IVA(OP1);
G2(OP2-) = IVA(OP2);
G3(OP3-) = IVA(OP3);
G4(S0) = ND2(OP3-,OP1-);
G5(S1) = ND2(OP2-,OP1-);
G6(S2) = ND3(OP3,OP1,OP0-);
G7(S3) = ND3(OP3,OP2,OP1);
G8(Z) = ND4(S0,S1,S2,S3);
END MODULE;

MODULE FANOUT-5;
INPUTS A;
OUTPUTS Z0,Z1,Z2,Z3,Z4;
LEVEL FUNCTION;
DEFINE
AA(AA) = B-BUF(A);
G0(Z0) = ID(AA);
```

```
G1(Z1) = ID(AA);
G2(Z2) = ID(AA);
G3(Z3) = ID(AA);
G4(Z4) = ID(AA);
END MODULE;

MODULE V-OR_5;
INPUTS A.0,A.1,A.2,A.3,A.4,B.0,B.1,B.2,B.3,B.4;
OUTPUTS Y.0,Y.1,Y.2,Y.3,Y.4;
LEVEL FUNCTION;
DEFINE
G.0(Y.0) = OR2(A.0,B.0);
G.1(Y.1) = OR2(A.1,B.1);
G.2(Y.2) = OR2(A.2,B.2);
G.3(Y.3) = OR2(A.3,B.3);
G.4(Y.4) = OR2(A.4,B.4);
END MODULE;

MODULE DECODE-5;
INPUTS S0,S1,S2,S3,S4;
OUTPUTS X00,X10,X20,X30,X01,X11,X21,X31,X02,X12,X22,X32,X03,X13,X23,X33,X04,
        X14,X24,X34,X05,X15,X25,X35,X06,X16,X26,X36,X07,X17,X27,X37;
LEVEL FUNCTION;
DEFINE
GS0-(S0-) = IVA(S0);
GS1-(S1-) = IVA(S1);
GS2-(S2-) = IVA(S2);
GS3-(S3-) = IVA(S3);
GS4-(S4-) = IVA(S4);
GS0(BS0) = IVA(S0-);
GS1(BS1) = IVA(S1-);
GS2(BS2) = IVA(S2-);
GS3(BS3) = IVA(S3-);
GS4(BS4) = IVA(S4-);
GL0(L0) = ND2(S0-,S1-);
GL1(L1) = ND2(BS0,S1-);
GL2(L2) = ND2(S0-,BS1);
GL3(L3) = ND2(BS0,BS1);
GH0(H0) = ND3(S2-,S3-,S4-);
GH1(H1) = ND3(BS2,S3-,S4-);
GH2(H2) = ND3(S2-,BS3,S4-);
GH3(H3) = ND3(BS2,BS3,S4-);
GH4(H4) = ND3(S2-,S3-,BS4);
GH5(H5) = ND3(BS2,S3-,BS4);
GH6(H6) = ND3(S2-,BS3,BS4);
GH7(H7) = ND3(BS2,BS3,BS4);
GX00(X00) = NR2(L0,H0);
GX10(X10) = NR2(L1,H0);
GX20(X20) = NR2(L2,H0);
GX30(X30) = NR2(L3,H0);
GX01(X01) = NR2(L0,H1);
GX11(X11) = NR2(L1,H1);
GX21(X21) = NR2(L2,H1);
GX31(X31) = NR2(L3,H1);
GX02(X02) = NR2(L0,H2);
```

```
GX12(X12) = NR2(L1,H2);
GX22(X22) = NR2(L2,H2);
GX32(X32) = NR2(L3,H2);
GX03(X03) = NR2(L0,H3);
GX13(X13) = NR2(L1,H3);
GX23(X23) = NR2(L2,H3);
GX33(X33) = NR2(L3,H3);
GX04(X04) = NR2(L0,H4);
GX14(X14) = NR2(L1,H4);
GX24(X24) = NR2(L2,H4);
GX34(X34) = NR2(L3,H4);
GX05(X05) = NR2(L0,H5);
GX15(X15) = NR2(L1,H5);
GX25(X25) = NR2(L2,H5);
GX35(X35) = NR2(L3,H5);
GX06(X06) = NR2(L0,H6);
GX16(X16) = NR2(L1,H6);
GX26(X26) = NR2(L2,H6);
GX36(X36) = NR2(L3,H6);
GX07(X07) = NR2(L0,H7);
GX17(X17) = NR2(L1,H7);
GX27(X27) = NR2(L2,H7);
GX37(X37) = NR2(L3,H7);
END MODULE;

MODULE NEXT-STATE;
INPUTS S.0,S.1,S.2,S.3,S.4,S.5,S.6,S.7,S.8,S.9,S.10,S.11,S.12,S.13,S.14,S.15,
       S.16,S.17,S.18,S.19,S.20,S.21,S.22,S.23,S.24,S.25,S.26,S.27,S.28,S.29,
       S.30,S.31,STORE,SET-SOME-FLAGS,UNARY,DIRECT-A,DIRECT-B,SIDE-EFFECT-A,
       SIDE-EFFECT-B,ALL-T-REGS-ADDRESS,DTACK-,HOLD-;
OUTPUTS W-17,W-18,W-19,W-22,W-24,W-27,W-29,W-44,W-47,W-48,W-49,W-52,W-61,
        W-62,W-63,W-66,W-74,W-75,W-76,W-79,W-82,W-83,W-89,W-93,W-98,W-99,
        W-112,W-113,W-105,W-106,W-110,W-111;
LEVEL FUNCTION;
DEFINE
R-0(FETCH0) = ID(S.0);
R-1(FETCH1) = ID(S.1);
R-2(FETCH2) = ID(S.2);
R-3(FETCH3) = ID(S.3);
R-4(DECODE) = ID(S.4);
R-5(REGA) = ID(S.5);
R-6(REGB) = ID(S.6);
R-7(UPDATE) = ID(S.7);
R-8(READA0) = ID(S.8);
R-9(READA1) = ID(S.9);
R-10(READA2) = ID(S.10);
R-11(READA3) = ID(S.11);
R-12(READB0) = ID(S.12);
R-13(READB1) = ID(S.13);
R-14(READB2) = ID(S.14);
R-15(READB3) = ID(S.15);
R-16(WRITE0) = ID(S.16);
R-17(WRITE1) = ID(S.17);
R-18(WRITE2) = ID(S.18);
R-19(WRITE3) = ID(S.19);
```

```
R-20(SEFA0) = ID(S.20);
R-21(SEFA1) = ID(S.21);
R-22(SEFB0) = ID(S.22);
R-23(SEFB1) = ID(S.23);
R-24(HOLD0) = ID(S.24);
R-25(HOLD1) = ID(S.25);
R-26(V11010) = ID(S.26);
R-27(V11011) = ID(S.27);
R-28(RESET0) = ID(S.28);
R-29(RESET1) = ID(S.29);
R-30(RESET2) = ID(S.30);
R-31(V11111) = ID(S.31);
G-111(W-111) = VSS();
G-109(W-109) = IVA(RESET1);
G-107(W-107) = IVA(ALL-T-REGS-ADDRESS);
G-108(W-108) = ND2(W-107,RESET2);
G-110(W-110) = ND2(W-108,W-109);
G-106(W-106) = ID(RESET0);
G-104(W-104) = IVA(V11010);
G-103(W-103) = IVA(V11011);
G-102(W-102) = IVA(V11111);
G-105(W-105) = ND3(W-102,W-103,W-104);
G-99(W-99) = AN2(HOLD-,HOLD0);
G-96(W-96) = IVA(HOLD-);
G-97(W-97) = ND2(W-96,FETCH1);
G-95(W-95) = ND2(W-96,HOLD0);
G-98(W-98) = ND2(W-95,W-97);
G-91(W-91) = AN2(SIDE-EFFECT-B,UNARY);
G-92(W-92) = ND2(W-91,UPDATE);
G-90(W-90) = IVA(SEFB0);
G-93(W-93) = ND2(W-90,W-92);
G-86(W-86) = OR2(STORE,SET-SOME-FLAGS);
G-87(W-87) = IVA(W-86);
G-85(W-85) = IVA(SIDE-EFFECT-A);
G-88(W-88) = ND4(SIDE-EFFECT-B,W-85,W-87,DECODE);
G-84(W-84) = ND2(SIDE-EFFECT-B,SEFA1);
G-89(W-89) = ND2(W-84,W-88);
G-83(W-83) = ID(SEFA0);
G-82(W-82) = AN3(SIDE-EFFECT-A,W-87,DECODE);
G-78(W-78) = IVA(WRITE2);
G-77(W-77) = ND2(DTACK-,WRITE3);
G-79(W-79) = ND2(W-77,W-78);
G-76(W-76) = ID(WRITE1);
G-75(W-75) = ID(WRITE0);
G-72(W-72) = IVA(DIRECT-B);
G-73(W-73) = ND3(STORE,W-72,REGA);
G-70(W-70) = IVA(DTACK-);
G-71(W-71) = ND5(STORE,UNARY,W-72,W-70,READA3);
G-68(W-68) = ND3(STORE,W-70,READB3);
G-74(W-74) = ND3(W-68,W-71,W-73);
G-65(W-65) = IVA(READB2);
G-64(W-64) = ND2(DTACK-,READB3);
G-66(W-66) = ND2(W-64,W-65);
G-63(W-63) = ID(READB1);
G-62(W-62) = ID(READB0);
```

```
G-57(W-57) = OR2(DIRECT-B,UNARY);
G-58(W-58) = IVA(W-57);
G-60(W-60) = ND4(W-58,DIRECT-A,W-86,DECODE);
G-53(W-53) = IVA(UNARY);
G-56(W-56) = ND4(W-53,W-72,W-70,READA3);
G-61(W-61) = ND2(W-56,W-60);
G-51(W-51) = IVA(READA2);
G-50(W-50) = ND2(DTACK-,READA3);
G-52(W-52) = ND2(W-50,W-51);
G-49(W-49) = ID(READA1);
G-48(W-48) = ID(READA0);
G-45(W-45) = IVA(DIRECT-A);
G-47(W-47) = AN3(W-45,W-86,DECODE);
G-43(W-43) = ND3(UNARY,DIRECT-B,REGA);
G-40(W-40) = IVA(STORE);
G-42(W-42) = ND3(W-40,W-72,REGA);
G-39(W-39) = IVA(REGB);
G-38(W-38) = ND3(DIRECT-B,W-70,READA3);
G-36(W-36) = ND5(W-40,UNARY,W-72,W-70,READA3);
G-32(W-32) = ND3(W-40,W-70,READB3);
G-44(W-44) = ND6(W-32,W-36,W-38,W-39,W-42,W-43);
G-29(W-29) = AN3(W-53,DIRECT-B,REGA);
G-27(W-27) = AN4(W-57,DIRECT-A,W-86,DECODE);
G-24(W-24) = AN2(W-70,FETCH3);
G-21(W-21) = IVA(FETCH2);
G-20(W-20) = ND2(DTACK-,FETCH3);
G-22(W-22) = ND2(W-20,W-21);
G-19(W-19) = AN2(HOLD-,FETCH1);
G-18(W-18) = ID(FETCH0);
G-11(W-11) = IVA(SIDE-EFFECT-B);
G-15(W-15) = ND4(W-11,W-85,W-87,DECODE);
G-9(W-9) = IVA(W-91);
G-10(W-10) = ND2(W-9,UPDATE);
G-7(W-7) = ND2(W-70,WRITE3);
G-16(W-16) = AN3(W-7,W-10,W-15);
G-4(W-4) = ND2(W-11,SEFA1);
G-2(W-2) = IVA(SEFB1);
G-1(W-1) = IVA(HOLD1);
G-0(W-0) = ND2(ALL-T-REGS-ADDRESS,RESET2);
G-5(W-5) = AN4(W-0,W-1,W-2,W-4);
G-17(W-17) = ND2(W-5,W-16);
G-112(W-112) = ID(W-111);
G-113(W-113) = ID(W-111);
END MODULE;

MODULE CV;
INPUTS DECODED-STATE.0,DECODED-STATE.1,DECODED-STATE.2,DECODED-STATE.3,
       DECODED-STATE.4,DECODED-STATE.5,DECODED-STATE.6,DECODED-STATE.7,
       DECODED-STATE.8,DECODED-STATE.9,DECODED-STATE.10,DECODED-STATE.11,
       DECODED-STATE.12,DECODED-STATE.13,DECODED-STATE.14,DECODED-STATE.15,
       DECODED-STATE.16,DECODED-STATE.17,DECODED-STATE.18,DECODED-STATE.19,
       DECODED-STATE.20,DECODED-STATE.21,DECODED-STATE.22,DECODED-STATE.23,
       DECODED-STATE.24,DECODED-STATE.25,DECODED-STATE.26,DECODED-STATE.27,
       DECODED-STATE.28,DECODED-STATE.29,DECODED-STATE.30,DECODED-STATE.31,
       RN-A.0,RN-A.1,RN-A.2,RN-A.3,RN-B.0,RN-B.1,RN-B.2,RN-B.3,OP-CODE.0,
```

```
       OP-CODE.1,OP-CODE.2,OP-CODE.3,PC-REG.0,PC-REG.1,PC-REG.2,PC-REG.3,
       REGS-ADDRESS.0,REGS-ADDRESS.1,REGS-ADDRESS.2,REGS-ADDRESS.3,
       SET-FLAGS.0,SET-FLAGS.1,SET-FLAGS.2,SET-FLAGS.3,STORE,C,A-IMMEDIATE-P,
       RW-,DIRECT-A,SIDE-EFFECT-A,SIDE-EFFECT-B,PRE-DEC-A,PRE-DEC-B;
OUTPUTS NEW-RW-,STROBE-,HDACK-,WE-REGS,WE-A-REG,WE-B-REG,WE-I-REG,
        WE-DATA-OUT,WE-ADDR-OUT,WE-HOLD-,WE-PC-REG,DATA-IN-SELECT,
        DEC-ADDR-OUT,SELECT-IMMEDIATE,ALU-C,ALU-ZERO,STATE.0,STATE.1,STATE.2,
        STATE.3,STATE.4,WE-FLAGS.0,WE-FLAGS.1,WE-FLAGS.2,WE-FLAGS.3,
        NEW-REGS-ADDRESS.0,NEW-REGS-ADDRESS.1,NEW-REGS-ADDRESS.2,
        NEW-REGS-ADDRESS.3,ALU-OP.0,ALU-OP.1,ALU-OP.2,ALU-OP.3,ALU-MPG.0,
        ALU-MPG.1,ALU-MPG.2,ALU-MPG.3,ALU-MPG.4,ALU-MPG.5,ALU-MPG.6;
LEVEL FUNCTION;
DEFINE
G-FETCH0(FETCH0) = ID(DECODED-STATE.0);
G-FETCH1(FETCH1) = ID(DECODED-STATE.1);
G-FETCH2(FETCH2) = ID(DECODED-STATE.2);
G-FETCH3(FETCH3) = ID(DECODED-STATE.3);
G-DECODE(DECODE) = ID(DECODED-STATE.4);
G-REGA(REGA) = ID(DECODED-STATE.5);
G-REGB(REGB) = ID(DECODED-STATE.6);
G-UPDATE(UPDATE) = ID(DECODED-STATE.7);
G-READA0(READA0) = ID(DECODED-STATE.8);
G-READA1(READA1) = ID(DECODED-STATE.9);
G-READA2(READA2) = ID(DECODED-STATE.10);
G-READA3(READA3) = ID(DECODED-STATE.11);
G-READB0(READB0) = ID(DECODED-STATE.12);
G-READB1(READB1) = ID(DECODED-STATE.13);
G-READB2(READB2) = ID(DECODED-STATE.14);
G-READB3(READB3) = ID(DECODED-STATE.15);
G-WRITE0(WRITE0) = ID(DECODED-STATE.16);
G-WRITE1(WRITE1) = ID(DECODED-STATE.17);
G-WRITE2(WRITE2) = ID(DECODED-STATE.18);
G-WRITE3(WRITE3) = ID(DECODED-STATE.19);
G-SEFA0(SEFA0) = ID(DECODED-STATE.20);
G-SEFA1(SEFA1) = ID(DECODED-STATE.21);
G-SEFB0(SEFB0) = ID(DECODED-STATE.22);
G-SEFB1(SEFB1) = ID(DECODED-STATE.23);
G-HOLD0(HOLD0) = ID(DECODED-STATE.24);
G-HOLD1(HOLD1) = ID(DECODED-STATE.25);
G-V11010(V11010) = ID(DECODED-STATE.26);
G-V11011(V11011) = ID(DECODED-STATE.27);
G-RESET0(RESET0) = ID(DECODED-STATE.28);
G-RESET1(RESET1) = ID(DECODED-STATE.29);
G-RESET2(RESET2) = ID(DECODED-STATE.30);
G-V11111(V11111) = ID(DECODED-STATE.31);
G0(STORE-) = IVA(STORE);
G1(ALU-ZERO) = OR4(FETCH0,RESET0,RESET1,RESET2);
G2(ALU-SWAP) = OR3(READB2,WRITE1,SEFB1);
G3(INCDECA) = OR2(READA1,SEFA1);
G4(S4) = NR4(FETCH2,INCDECA,ALU-SWAP,ALU-ZERO);
G5(S5) = ND2(INCDECA,PRE-DEC-A);
G6(S6) = ND2(ALU-SWAP,PRE-DEC-B);
G7(S7) = ND2(S5,S6);
G8(ALU-OP.0,ALU-OP.1,ALU-OP.2,ALU-OP.3)
  = SELECT-OP-CODE(S4,S7,OP-CODE.0,OP-CODE.1,OP-CODE.2,OP-CODE.3);
```

```
G10(S10) = IVA(RW-);
G11(S11) = ND2(S10,FETCH0);
G12(S12) = NR3(WRITE1,WRITE2,WRITE3);
G13(NEW-RW-) = AN2(S11,S12);
G14(STROBE-) = NR8(FETCH2,FETCH3,READA2,READA3,READB2,READB3,WRITE2,WRITE3);
G15(HDACK-) = IVA(HOLD0);
G17(S17) = ND2(STORE,UPDATE);
G18(S18) = ND2(SIDE-EFFECT-A,READA1);
G19(S19) = ND3(STORE-,SIDE-EFFECT-B,READB2);
G20(S20) = ND2(SIDE-EFFECT-B,WRITE1);
G21(S21) = NR5(FETCH2,SEFA1,SEFB1,RESET0,RESET2);
G22(WE-REGS) = ND5(S17,S18,S19,S20,S21);
G23(S23) = ND2(DIRECT-A,READB1);
G24(S24) = NR6(FETCH0,REGA,READA0,READA3,SEFA0,RESET1);
G25(WE-A-REG) = ND2(S23,S24);
G26(S26) = NR4(REGB,UPDATE,READA2,READB0);
G27(S27) = NR4(READB3,WRITE0,SEFB0,RESET1);
G28(WE-B-REG) = ND2(S26,S27);
G29(WE-I-REG) = OR2(FETCH3,RESET1);
G30(WE-DATA-OUT) = OR2(WRITE0,RESET0);
G31(S31) = NR3(FETCH0,READA0,READB0);
G32(S32) = NR2(WRITE0,RESET1);
G33(WE-ADDR-OUT) = ND2(S31,S32);
G34(WE-HOLD-) = OR3(FETCH0,HOLD0,RESET0);
G35(WE-PC-REG) = OR2(HOLD0,RESET0);
G36(DATA-IN-SELECT) = OR3(FETCH3,READA3,READB3);
G37(S37) = ND2(PRE-DEC-A,READA0);
G38(S38) = OR2(READB0,WRITE0);
G39(S39) = ND2(PRE-DEC-B,S38);
G40(DEC-ADDR-OUT) = ND2(S37,S39);
G41(S41) = OR2(REGA,READB1);
G42(SELECT-IMMEDIATE) = AN2(A-IMMEDIATE-P,S41);
G43(ALU-C) = CARRY-IN-HELP(C,ALU-ZERO,ALU-OP.0,ALU-OP.1,ALU-OP.2,ALU-OP.3);
G44(STATE.0,STATE.1,STATE.2,STATE.3,STATE.4)
  = ENCODE-32(DECODED-STATE.0,DECODED-STATE.1,DECODED-STATE.2,
              DECODED-STATE.3,DECODED-STATE.4,DECODED-STATE.5,
              DECODED-STATE.6,DECODED-STATE.7,DECODED-STATE.8,
              DECODED-STATE.9,DECODED-STATE.10,DECODED-STATE.11,
              DECODED-STATE.12,DECODED-STATE.13,DECODED-STATE.14,
              DECODED-STATE.15,DECODED-STATE.16,DECODED-STATE.17,
              DECODED-STATE.18,DECODED-STATE.19,DECODED-STATE.20,
              DECODED-STATE.21,DECODED-STATE.22,DECODED-STATE.23,
              DECODED-STATE.24,DECODED-STATE.25,DECODED-STATE.26,
              DECODED-STATE.27,DECODED-STATE.28,DECODED-STATE.29,
              DECODED-STATE.30,DECODED-STATE.31);
G45(FANOUT-RESET0.0,FANOUT-RESET0.1,FANOUT-RESET0.2,FANOUT-RESET0.3)
  = FANOUT-4(RESET0);
G46(S46) = NR2(UPDATE,WRITE0);
G47(WE-FLAGS.0,WE-FLAGS.1,WE-FLAGS.2,WE-FLAGS.3)
  = TV-IF_4(S46,FANOUT-RESET0.0,FANOUT-RESET0.1,FANOUT-RESET0.2,
           FANOUT-RESET0.3,SET-FLAGS.0,SET-FLAGS.1,SET-FLAGS.2,SET-FLAGS.3);
G48(S48) = NR3(REGA,READA0,READA1);
G49(S49) = NR3(READB1,SEFA0,SEFA1);
G50(SELECT-RN-A) = ND2(S48,S49);
G51(S51) = NR5(REGB,UPDATE,READA2,READB0,READB2);
```

```
G52(S52) = NR4(WRITE0,WRITE1,SEFB0,SEFB1);
G53(SELECT-RN-B) = ND2(S51,S52);
G54(SELECT-ALL-F) = OR2(RESET0,RESET1);
G55(V-INC-REGS-ADDRESS.0,V-INC-REGS-ADDRESS.1,V-INC-REGS-ADDRESS.2,
    V-INC-REGS-ADDRESS.3)
  = V-INC4(REGS-ADDRESS.0,REGS-ADDRESS.1,REGS-ADDRESS.2,REGS-ADDRESS.3);
G56(S56.0,S56.1,S56.2,S56.3)
  = TV-IF_4(RESET2,V-INC-REGS-ADDRESS.0,V-INC-REGS-ADDRESS.1,
            V-INC-REGS-ADDRESS.2,V-INC-REGS-ADDRESS.3,PC-REG.0,PC-REG.1,
            PC-REG.2,PC-REG.3);
G57(S57.0,S57.1,S57.2,S57.3) = V-IF-F-4(SELECT-ALL-F,S56.0,S56.1,S56.2,S56.3);
G58(S58.0,S58.1,S58.2,S58.3)
  = TV-IF_4(SELECT-RN-B,RN-B.0,RN-B.1,RN-B.2,RN-B.3,S57.0,S57.1,S57.2,S57.3);
G59(NEW-REGS-ADDRESS.0,NEW-REGS-ADDRESS.1,NEW-REGS-ADDRESS.2,
    NEW-REGS-ADDRESS.3)
  = TV-IF_4(SELECT-RN-A,RN-A.0,RN-A.1,RN-A.2,RN-A.3,S58.0,S58.1,S58.2,S58.3);
G60(ALU-MPG.0,ALU-MPG.1,ALU-MPG.2,ALU-MPG.3,ALU-MPG.4,ALU-MPG.5,ALU-MPG.6)
  = MPG(ALU-ZERO,ALU-SWAP,ALU-OP.0,ALU-OP.1,ALU-OP.2,ALU-OP.3);
END MODULE;

MODULE SELECT-OP-CODE;
INPUTS SELECT,DEC,OP0,OP1,OP2,OP3;
OUTPUTS Z0,Z1,Z2,Z3;
LEVEL FUNCTION;
DEFINE
I0(OP0-) = IVA(OP0);
G0(Z0) = ND2(SELECT,OP0-);
G1(Z1) = AN2(SELECT,OP1);
G2(Z2) = MUX21H(DEC,OP2,SELECT);
G3(Z3) = AN2(SELECT,OP3);
END MODULE;

MODULE TV-IF_4;
INPUTS C,A.0,A.1,A.2,A.3,B.0,B.1,B.2,B.3;
OUTPUTS OUT.0,OUT.1,OUT.2,OUT.3;
LEVEL FUNCTION;
DEFINE
C-BUF(C-BUF) = B-BUF(C);
LEFT(OUT.0,OUT.1) = TV-IF_2(C-BUF,A.0,A.1,B.0,B.1);
RIGHT(OUT.2,OUT.3) = TV-IF_2(C-BUF,A.2,A.3,B.2,B.3);
END MODULE;

MODULE TV-IF_2;
INPUTS C,A.0,A.1,B.0,B.1;
OUTPUTS OUT.0,OUT.1;
LEVEL FUNCTION;
DEFINE
LEFT(OUT.0) = TV-IF_1(C,A.0,B.0);
RIGHT(OUT.1) = TV-IF_1(C,A.1,B.1);
END MODULE;

MODULE TV-IF_1;
INPUTS C,A.0,B.0;
OUTPUTS OUT.0;
LEVEL FUNCTION;
```

```
DEFINE
LEAF(OUT.0) = MUX21H(B.0,A.0,C);
END MODULE;

MODULE CARRY-IN-HELP;
INPUTS CIN,Z,OP0IN,OP1IN,OP2IN,OP3IN;
OUTPUTS COUT;
LEVEL FUNCTION;
DEFINE
G0(C-,C) = IVDA(CIN);
G1(OP0-,OP0) = IVDA(OP0IN);
G2(OP1-,OP1) = IVDA(OP1IN);
G3(OP2-,OP2) = IVDA(OP2IN);
G4(OP3-,OP3) = IVDA(OP3IN);
G5(S5) = ND3(OP1-,OP2-,OP3-);
G6(S6) = ND3(OP0-,OP1-,OP2);
G7(S7) = ND3(OP0,OP1,OP2);
G8(S8) = ND3(S5,S6,S7);
G9(S9) = ND2(OP3,C);
G10(S10) = ND3(OP0-,OP2-,C);
G11(S11) = ND3(OP0-,OP2,C-);
G12(S12) = ND3(S9,S10,S11);
G13(COUT) = OR2(S8,S12);
END MODULE;

MODULE ENCODE-32;
INPUTS S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16,S17,S18,S19,
       S20,S21,S22,S23,S24,S25,S26,S27,S28,S29,S30,S31;
OUTPUTS W-4,W-9,W-14,W-19,W-24;
LEVEL FUNCTION;
DEFINE
G-23(W-23) = NR3(S28,S29,S30);
G-22(W-22) = NR2(S24,S25);
G-21(W-21) = NR4(S20,S21,S22,S23);
G-20(W-20) = NR4(S16,S17,S18,S19);
G-24(W-24) = ND4(W-20,W-21,W-22,W-23);
G-16(W-16) = NR4(S12,S13,S14,S15);
G-15(W-15) = NR4(S8,S9,S10,S11);
G-19(W-19) = ND4(W-15,W-16,W-22,W-23);
G-10(W-10) = NR4(S4,S5,S6,S7);
G-14(W-14) = ND4(W-10,W-16,W-21,W-23);
G-8(W-8) = IVA(S30);
G-7(W-7) = NR4(S18,S19,S22,S23);
G-6(W-6) = NR4(S10,S11,S14,S15);
G-5(W-5) = NR4(S2,S3,S6,S7);
G-9(W-9) = ND4(W-5,W-6,W-7,W-8);
G-3(W-3) = NR2(S25,S29);
G-2(W-2) = NR4(S17,S19,S21,S23);
G-1(W-1) = NR4(S9,S11,S13,S15);
G-0(W-0) = NR4(S1,S3,S5,S7);
G-4(W-4) = ND4(W-0,W-1,W-2,W-3);
END MODULE;

MODULE FANOUT-4;
INPUTS A;
```

```
OUTPUTS Z0,Z1,Z2,Z3;
LEVEL FUNCTION;
DEFINE
AA(AA) = B-BUF(A);
G0(Z0) = ID(AA);
G1(Z1) = ID(AA);
G2(Z2) = ID(AA);
G3(Z3) = ID(AA);
END MODULE;

MODULE V-INC4;
INPUTS A0,A1,A2,A3;
OUTPUTS W-10,W-3,W-8,W-14;
LEVEL FUNCTION;
DEFINE
G-12(W-12) = IVA(A2);
G-11(W-11) = IVA(A1);
G-10(W-10) = IVA(A0);
G-13(W-13) = NR3(W-10,W-11,W-12);
G-9(W-9) = IVA(A3);
G-14(W-14) = EN(W-9,W-13);
G-7(W-7) = NR2(W-10,W-11);
G-8(W-8) = EN(W-12,W-7);
G-3(W-3) = EO(W-11,W-10);
END MODULE;

MODULE V-IF-F-4;
INPUTS C,A0,A1,A2,A3;
OUTPUTS Z0,Z1,Z2,Z3;
LEVEL FUNCTION;
DEFINE
CB(C-) = IVA(C);
G0(Z0) = AN2(C-,A0);
G1(Z1) = AN2(C-,A1);
G2(Z2) = AN2(C-,A2);
G3(Z3) = AN2(C-,A3);
END MODULE;

MODULE MPG;
INPUTS ZERO,SWAP,OP0,OP1,OP2,OP3;
OUTPUTS GBN,GAN,GA,PB,PAN,PA,MODE;
LEVEL FUNCTION;
DEFINE
M(MODE) = DECODE-MODE(OP0,OP1,OP2,OP3);
P(PB,PAN,PA) = DECODE-PROP(ZERO,SWAP,OP0,OP1,OP2,OP3);
G(GBN,GAN,GA) = DECODE-GEN(ZERO,SWAP,OP0,OP1,OP2,OP3);
END MODULE;

MODULE DECODE-MODE;
INPUTS OP0,OP1,OP2,OP3;
OUTPUTS W-1;
LEVEL FUNCTION;
DEFINE
G-0(W-0) = NR3(OP0,OP1,OP2);
G-1(W-1) = NR2(W-0,OP3);
```

```
END MODULE;

MODULE DECODE-PROP;
INPUTS ZERO,SWAP,OP0,OP1,OP2,OP3;
OUTPUTS W-16,W-24,W-48;
LEVEL FUNCTION;
DEFINE
G-47(W-47) = IVA(ZERO);
G-44(W-44) = IVA(OP3);
G-42(W-42) = IVA(OP2);
G-43(W-43) = IVA(W-42);
G-41(W-41) = IVA(OP1);
G-39(W-39) = IVA(OP0);
G-40(W-40) = IVA(W-39);
G-45(W-45) = ND4(W-40,W-41,W-43,W-44);
G-34(W-34) = IVA(SWAP);
G-35(W-35) = IVA(W-34);
G-37(W-37) = ND2(W-35,W-44);
G-38(W-38) = ND2(W-42,W-37);
G-30(W-30) = IVA(W-41);
G-31(W-31) = EN(W-40,W-30);
G-26(W-26) = IVA(W-44);
G-32(W-32) = ND2(W-26,W-31);
G-46(W-46) = ND3(W-32,W-38,W-45);
G-48(W-48) = AN2(W-46,W-47);
G-22(W-22) = NR2(W-40,W-41);
G-23(W-23) = NR2(W-44,W-22);
G-24(W-24) = NR2(W-42,W-23);
G-15(W-15) = ND3(W-42,W-44,W-35);
G-10(W-10) = ND2(W-30,W-44);
G-6(W-6) = ND4(W-39,W-41,W-43,W-26);
G-16(W-16) = ND3(W-6,W-10,W-15);
END MODULE;

MODULE DECODE-GEN;
INPUTS ZERO,SWAP,OP0,OP1,OP2,OP3;
OUTPUTS W-22,W-41,W-66;
LEVEL FUNCTION;
DEFINE
G-64(W-64) = IVA(ZERO);
G-65(W-65) = IVA(W-64);
G-61(W-61) = IVA(OP3);
G-60(W-60) = IVA(OP2);
G-58(W-58) = IVA(OP1);
G-59(W-59) = IVA(W-58);
G-62(W-62) = ND3(W-59,W-60,W-61);
G-56(W-56) = IVA(W-60);
G-52(W-52) = IVA(OP0);
G-53(W-53) = IVA(W-52);
G-57(W-57) = ND3(W-53,W-58,W-56);
G-50(W-50) = EO(W-59,W-56);
G-45(W-45) = IVA(W-61);
G-51(W-51) = ND3(W-53,W-45,W-50);
G-63(W-63) = ND3(W-51,W-57,W-62);
G-66(W-66) = NR2(W-63,W-65);
```

```
G-35(W-35) = IVA(SWAP);
G-36(W-36) = ND2(W-58,W-35);
G-37(W-37) = ND3(W-56,W-61,W-36);
G-30(W-30) = ND4(W-53,W-59,W-60,W-45);
G-38(W-38) = ND2(W-30,W-37);
G-41(W-41) = NR2(W-38,W-65);
G-22(W-22) = ND3(W-51,W-37,W-62);
END MODULE;

MODULE TTL-INPUT-PADS_4;
INPUTS PI,IN.0,IN.1,IN.2,IN.3;
OUTPUTS PO.3,B-OUT.0,B-OUT.1,B-OUT.2,B-OUT.3;
LEVEL FUNCTION;
DEFINE
G.0(OUT.0,PO.0) = &TLCHT(IN.0,PI);
B.0(B-OUT.0) = B-BUF(OUT.0);
G.1(OUT.1,PO.1) = &TLCHT(IN.1,PO.0);
B.1(B-OUT.1) = B-BUF(OUT.1);
G.2(OUT.2,PO.2) = &TLCHT(IN.2,PO.1);
B.2(B-OUT.2) = B-BUF(OUT.2);
G.3(OUT.3,PO.3) = &TLCHT(IN.3,PO.2);
B.3(B-OUT.3) = B-BUF(OUT.3);
END MODULE;

MODULE TTL-TRI-OUTPUT-PADS_32;
INPUTS ENABLE,IN.0,IN.1,IN.2,IN.3,IN.4,IN.5,IN.6,IN.7,IN.8,IN.9,IN.10,IN.11,
       IN.12,IN.13,IN.14,IN.15,IN.16,IN.17,IN.18,IN.19,IN.20,IN.21,IN.22,
       IN.23,IN.24,IN.25,IN.26,IN.27,IN.28,IN.29,IN.30,IN.31;
OUTPUTS OUT.0,OUT.1,OUT.2,OUT.3,OUT.4,OUT.5,OUT.6,OUT.7,OUT.8,OUT.9,OUT.10,
        OUT.11,OUT.12,OUT.13,OUT.14,OUT.15,OUT.16,OUT.17,OUT.18,OUT.19,
        OUT.20,OUT.21,OUT.22,OUT.23,OUT.24,OUT.25,OUT.26,OUT.27,OUT.28,
        OUT.29,OUT.30,OUT.31;
LEVEL FUNCTION;
DEFINE
ENABLE-BUFFER(ENABLE-BUF) = B-BUF-PWR(ENABLE);
G.0(OUT.0) = &BT8RP(IN.0,ENABLE-BUF);
G.1(OUT.1) = &BT8RP(IN.1,ENABLE-BUF);
G.2(OUT.2) = &BT8RP(IN.2,ENABLE-BUF);
G.3(OUT.3) = &BT8RP(IN.3,ENABLE-BUF);
G.4(OUT.4) = &BT8RP(IN.4,ENABLE-BUF);
G.5(OUT.5) = &BT8RP(IN.5,ENABLE-BUF);
G.6(OUT.6) = &BT8RP(IN.6,ENABLE-BUF);
G.7(OUT.7) = &BT8RP(IN.7,ENABLE-BUF);
G.8(OUT.8) = &BT8RP(IN.8,ENABLE-BUF);
G.9(OUT.9) = &BT8RP(IN.9,ENABLE-BUF);
G.10(OUT.10) = &BT8RP(IN.10,ENABLE-BUF);
G.11(OUT.11) = &BT8RP(IN.11,ENABLE-BUF);
G.12(OUT.12) = &BT8RP(IN.12,ENABLE-BUF);
G.13(OUT.13) = &BT8RP(IN.13,ENABLE-BUF);
G.14(OUT.14) = &BT8RP(IN.14,ENABLE-BUF);
G.15(OUT.15) = &BT8RP(IN.15,ENABLE-BUF);
G.16(OUT.16) = &BT8RP(IN.16,ENABLE-BUF);
G.17(OUT.17) = &BT8RP(IN.17,ENABLE-BUF);
G.18(OUT.18) = &BT8RP(IN.18,ENABLE-BUF);
G.19(OUT.19) = &BT8RP(IN.19,ENABLE-BUF);
```

```
G.20(OUT.20) = &BT8RP(IN.20,ENABLE-BUF);
G.21(OUT.21) = &BT8RP(IN.21,ENABLE-BUF);
G.22(OUT.22) = &BT8RP(IN.22,ENABLE-BUF);
G.23(OUT.23) = &BT8RP(IN.23,ENABLE-BUF);
G.24(OUT.24) = &BT8RP(IN.24,ENABLE-BUF);
G.25(OUT.25) = &BT8RP(IN.25,ENABLE-BUF);
G.26(OUT.26) = &BT8RP(IN.26,ENABLE-BUF);
G.27(OUT.27) = &BT8RP(IN.27,ENABLE-BUF);
G.28(OUT.28) = &BT8RP(IN.28,ENABLE-BUF);
G.29(OUT.29) = &BT8RP(IN.29,ENABLE-BUF);
G.30(OUT.30) = &BT8RP(IN.30,ENABLE-BUF);
G.31(OUT.31) = &BT8RP(IN.31,ENABLE-BUF);
END MODULE;

MODULE B-BUF-PWR;
INPUTS IN;
OUTPUTS OUT;
LEVEL FUNCTION;
DEFINE
G0(OUT-) = IVA(IN);
G1(OUT) = B4IP(OUT-);
END MODULE;

MODULE TTL-OUTPUT-PADS_4;
INPUTS IN.0,IN.1,IN.2,IN.3;
OUTPUTS OUT.0,OUT.1,OUT.2,OUT.3;
LEVEL FUNCTION;
DEFINE
B.0(B-IN.0) = B-BUF(IN.0);
G.0(OUT.0) = &B8RP(B-IN.0);
B.1(B-IN.1) = B-BUF(IN.1);
G.1(OUT.1) = &B8RP(B-IN.1);
B.2(B-IN.2) = B-BUF(IN.2);
G.2(OUT.2) = &B8RP(B-IN.2);
B.3(B-IN.3) = B-BUF(IN.3);
G.3(OUT.3) = &B8RP(B-IN.3);
END MODULE;

MODULE TTL-OUTPUT-PADS_5;
INPUTS IN.0,IN.1,IN.2,IN.3,IN.4;
OUTPUTS OUT.0,OUT.1,OUT.2,OUT.3,OUT.4;
LEVEL FUNCTION;
DEFINE
B.0(B-IN.0) = B-BUF(IN.0);
G.0(OUT.0) = &B8RP(B-IN.0);
B.1(B-IN.1) = B-BUF(IN.1);
G.1(OUT.1) = &B8RP(B-IN.1);
B.2(B-IN.2) = B-BUF(IN.2);
G.2(OUT.2) = &B8RP(B-IN.2);
B.3(B-IN.3) = B-BUF(IN.3);
G.3(OUT.3) = &B8RP(B-IN.3);
B.4(B-IN.4) = B-BUF(IN.4);
G.4(OUT.4) = &B8RP(B-IN.4);
END MODULE;
```

```
MODULE B-BUF;
INPUTS IN;
OUTPUTS OUT;
LEVEL FUNCTION;
DEFINE
G0(OUT-,OUT) = IVDA(IN);
END MODULE;

MODULE RAM-ENABLE-CIRCUIT;
INPUTS CLK,TEST-REGFILE-,DISABLE-REGFILE-,WE;
OUTPUTS Z;
LEVEL FUNCTION;
DEFINE
G0(CLK-10) = DEL10(CLK);
G1(TEST-REGFILE) = IVA(TEST-REGFILE-);
G2(GATE-CLK) = OR2(CLK-10,TEST-REGFILE);
G3(Z) = ND3P(WE,DISABLE-REGFILE-,GATE-CLK);
END MODULE;

MODULE ID;
INPUTS A;
OUTPUTS Z;
LEVEL FUNCTION;
DEFINE
Z = (A);
END MODULE;

MODULE VDD;
OUTPUTS Z;
LEVEL FUNCTION;
DEFINE
G0(Z) = ID(NC/1/);
END MODULE;

MODULE VSS;
OUTPUTS Z;
LEVEL FUNCTION;
DEFINE
G0(Z) = ID(NC/0/);
END MODULE;

END COMPILE;
END;
```

Index

Bibliography

[1] Kenneth L. Albin, Bishop C. Brock, Warren A. Hunt, Jr., and Lawrence M. Smith. Testing the FM9001 Microprocessor. Technical Report 90, Computational Logic, Inc., 1717 W. Sixth St., Suite 290, Austin, Texas 78703, 1994.

[2] William R. Bevier. *A Verified Operating System Kernel.* PhD Thesis, University of Texas at Austin, October 1987.

[3] W.R. Bevier. A Library for Hardware Verification. Draft Internal Note 57, Computational Logic, Inc., June 1988.

[4] W.R. Bevier. Kit and the Short Stack. *Journal of Automated Reasoning*, 5(4):519–530, December 1989.

[5] Robert S. Boyer and J Strother Moore. *A Computational Logic.* ACM Monograph Series. Academic Press, Inc., 1979.

[6] R.S. Boyer, D. Goldschlag, M. Kaufmann, and J S. Moore. Functional Instantiation in First Order Logic. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.

[7] R.S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

[8] Bishop Brock. Transforming Functional Hardware Models into Structural Models. Internal Note 150, Computational Logic, Inc., July 1989.

[9] Bishop C. Brock and Jr. Warren A. Hunt, Jr. The Formalization of a Simple HDL. In *Proceedings of the IFIP TC10/WG10.2/WG10.5 Workshop on Applied Formal Methods for Correct VLSI Design.* Elsevier Science Publishers, Amsterdam, 1990.

[10] Bishop C. Brock, Warren A. Hunt, Jr., and William D. Young. Introduction to a Formally Defined Hardware Description Language. *Theorem Provers in Circuit Design*, V. Stavridou, T. Melham, and R. Boute, eds., North-Holland, pp. 3–35, 1992.

[11] A. Camilleri, M. Gordon, and T. Melham. Hardware Verification Using Higher-order Logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 43–67. Elsevier Science Publishers, Amsterdam, 1987.

[12] M. Gordon. HOL: A Proof Generating System for Higher-order Logic. Technical Report 103, University of Cambridge, Computer Laboratory, 1987.

[13] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor*. LNCS 795, Springer-Verlag, 1994.

[14] Warren A. Hunt, Jr. Microprocessor Design Verification. *Journal of Automated Reasoning*, 5(4):429–460, December 1989.

[15] Warren A. Hunt, Jr. and Bishop Brock. A Formal HDL and Its Use in the FM9001 Verification. In C.A.R. Hoare and M.J.C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, pages 35–48. Prentice-Hall International Series in Computer Science, Englewood Cliffs, N.J., 1992.

[16] Matt Kaufmann. A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover. Technical Report CLI 19, Computational Logic, Inc., May 1988.

[17] Matt Kaufmann. An Integer Library for Nqthm. Internal Note 182, Computational Logic, Inc., March 1990.

[18] Matt Kaufmann. A Hardware Reset Lemma and Its Proof. Internal Note 230, Computational Logic, Inc., May 1991.

[19] J Strother Moore. A Mechanically Verified Language Implemenation. *Journal of Automated Reasoning*, 5(4):493–518, December 1989. Also published as CLI Technical Report 30.

[20] J Strother Moore. A Formal Model of Asynchronous Communication and Its Use in Mechanically Verifying a Biphase Mark Protocol. *Formal Aspects of Computing*, Vol. 6, No. 1, pp. 60–91, 1994.

[21] Lawrence M. Smith. FM9001 Model Validation on the LV500 Logic Verifier. Internal Report 299, Computational Logic, Inc., 1994.