Piton:
A Verified Assembly Level Language

by

J Strother Moore

Technical Report 22
September, 1988

This work was supported in part at Computational Logic, Inc., 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 and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.

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

Abstract

This report describes a programming language, its implementation on a microprocessor via a compiler, an assembler, and a linker, and the mechanically checked proof of the correctness of the implementation. The programming language, called Piton, is a high-level assembly language designed for verified applications and as the target language for high-level language compilers. It provides execute-only programs, recursive subroutine call and return, stack based parameter passing, local variables, global variables and arrays, a user-visible stack for intermediate results, and seven abstract data types including integers, data addresses, program addresses and subroutine names. Piton is formally specified by an interpreter written for it in the computational logic of Boyer and Moore. Piton has been implemented on the FM8502, a general purpose microprocessor whose gate-level design has been mechanically proved to implement its machine code interpreter. The FM8502 implementation of Piton is via a function in the Boyer-Moore logic which maps a Piton initial state into an FM8502 binary core image. The compiler, assembler and linker are all defined as functions in the logic. The implementation requires approximately 36K bytes and 1,400 lines of prettyprinted source code in the Pure Lisp-like syntax of the logic. The implementation has been mechanically proved correct. In particular, if a Piton state can be run to completion without error, then the final values of all the global data structures can be ascertained from an inspection of an FM8502 core image obtained by running the core image produced by the compiler, assembler, and linker. Thus, verified Piton programs running on FM8502 can be thought of as having been verified down to the gate level. The report defines Piton, exhibits a Piton program for big number addition and its correctness proof, describes the FM8502 implementation of Piton, formalizes the statement of correctness for the implementation, and describes the proof.

1. Introduction and Background

1.1. Motivation

The correctness of mechanically verified software typically rests on several informal assumptions about the correctness of the underlying subsystems. A correctly specified and accurately verified high-level language program may still exhibit incorrect behavior because of bugs in the compiler, assembler, linker, loader, runtime support software or the hardware. But, at least down to the hardware, these other components of the system are programs and hence are mathematical objects whose properties are subject to formal proof. It is the goal of *system verification* to specify and formally verify all of the components of a system down to the hardware level.

For our purposes it is convenient to define as software those components of a system accurately modelled by mathematical logic (as opposed to physics). Even so there is still a question of where to draw the line between software and hardware. What is the lowest level accurately modelled by mathematical logic? The machine code? The microcode? The register-transfer level? The layout and timing diagrams? Models of individual gates? A line must be drawn. The correctness of any physical system ultimately rests on statistically described assumptions about the properties of physical devices.

But because the software is accurately modelled by mathematical logic, and because mathematical proof is the ultimate arbiter of mathematical truth, the most reliable software systems will be those in which all the software components have been proved correct.

1.2. System Verification

The line has been drawn in a variety of places by various verification projects. The vast majority of verification work (as measured by funding levels) addresses ‘‘design proofs,’’ an activity in which the line is drawn somewhere *above* the highest level executable code in the system. Some verification work addresses ‘‘code proofs,’’ where traditionally the line has been drawn at the definition of a high-level programming language like Gypsy [7, 8, 9], Pascal [20], Fortran [3], and others [21, 6, 13, 17, 5]. There has been some work on compiler verification, notably the work of Polak [15] in which a compiler for a Pascal subset is verified. Finally, there has been some recent work closer to the bottom of the system stack. For example, Gordon [10] and Hunt [11] draw the line essentially at the register-transfer level and offer mechanically certified designs for digital hardware.

But the research underlying the construction of the first fully verified system must address more than just the verification of the individual components. The components of a system are built on top of each other. To verify a system one must be able to *stack* verified components. That is, one must be able to use what was proved about one level to construct the proof of the next.

This paper reports on what we believe is the first instance of stacking two verified components. In 1985 Warren Hunt designed, specified and proved the correctness of the FM8501 microprocessor. The FM8501 is a 16-bit, 8 register general purpose processor implementing a machine language with a conventional orthogonal instruction set. Hunt formally described the instruction set of the FM8501 by defining an interpreter for it in the computational logic designed by Boyer and Moore [2, 4]. Hunt also formally described the combinational logic and a register-transfer model that he claimed implemented the instruction set. The Boyer-Moore theorem prover was then used to prove that the register-transfer model correctly implemented the machine code interpreter.

The existence of a verified design for a general-purpose processor clearly suggests the idea of using that processor as the ‘‘delivery vehicle’’ for some ‘‘trusted system’’ such as an encryption box, a verifiable high-level language, or perhaps even a program verification system. This was an inevitable suggestion since FM8501 was developed in the same laboratory responsible for the Gypsy Verification Environment [9] and the Boyer-Moore theorem prover [2, 4], two of the most widely used program verification systems. But, unless one wants to build such tools in machine language, it is necessary to implement higher-level languages on FM8501. To maintain the credibility of the system, the implementation of those languages should be verified all the way down to the FM8501 machine code. Thus was born our ‘‘system verification’’ project.

1.3. The Piton Project

The first step was to design, implement and verify an assembly-level language on FM8501. We named the language ‘‘Piton’’ after the spikes driven into rocks by mountain climbers to secure their ropes.

When the Piton project began, the intended hardware base was FM8501. Early in the project we requested two changes to FM8501, which were implemented and verified by Warren Hunt. The modified machine was called the ‘‘FM8502.’’ The changes were (a) an increase in the word width from 16 to 32 bits and (b) the allocation of additional bits in the instruction word format to permit individual control over which of the alu condition code flags were stored. Because of the nature of the original FM8501 design, and the specification style, these changes were easy to make and to verify. Indeed, the FM8502 proof was produced from the FM8501 script without human assistance.

Before proceeding any further let it be clearly understood how far we are from having a verified system. FM8502 has not been physically realized. Building FM8502 while maintaining its credibility as a verified design would require much research. The ‘‘combinational logic’’ is described as an expression in propositional logic, not a gate graph in 3-space. Problems of layout, fanout, timing, etc., have not been addressed. The FM8502 design has no provision for testing the fabricated device. In short, because FM8502 is at the bottom of the stack, there is no tension (other than intellectual honesty and knowledge of engineering) to keep the underlying assumptions reasonable. Much can be done to further reduce our assumptions.

Even if built, FM8502 is inadequate as a practical delivery vehicle. It has no facilities for interrupts, memory management, or input/output. Even its instruction set is sometimes clumsy to use. Finally, FM8502 was originally designed as a PhD project to demonstrate the feasibility of hardware verification. The FM8501 design was very conventional (i.e., old fashioned) so that it was clear that the architectural style was not what made the design verifiable. Having made that point, we believe it would now be more cost effective to invest the effort to verify and build a more modern machine.

Another indication that we are far from having a verified system is that the Piton implementation described here is not an FM8502 program but a function in a computational logic. It can be executed (in Common Lisp) to generate an FM8502 core image. A goal not yet addressed by our research is how one might actually load that core image into a physically realized FM8502. A longer term goal is to make the compiler, assembler, and linker be FM8502 programs themselves. Consider, for example, an FM8502 machine code program **`impl`** that transforms Piton source code into an FM8502 core image that allegedly implements the Piton code. We believe that the best way to verify **`impl`** is to prove that its output is identical to that of a verified functionally defined implementation. This approach factors the correctness problem into three subgoals: (a) describe a suitable core image mathematically as a function on Piton source code, (b) prove that the mathematical function correctly implements the Piton code, and (c) prove that **impl** produces the described core image. The current work can thus be seen as the first two steps in producing a verified Piton compiler written as an FM8502 program.

These shortcomings notwithstanding, one must begin somewhere. The first fully verified system is unlikely to be built from perfectly suitable components constructed in one pass from perfectly suitable and credible hardware. No one has ever stacked two verified components of significant size before; we must learn how. FM8502 is sufficiently large and realistic to anchor one end of such a research project.

For the other end, we designed Piton, a simple assembly-level programming language. We wanted Piton's semantics to be relatively ''clean'' so that it was straightforward to prove properties of Piton programs. But the cleanliness could not come by making unrealistic assumptions, since it was our goal to implement the language and prove the implementation correct. Finally, we wanted the implementation to be efficient so that one could actually use Piton in verified applications. We imagine using Piton directly to write encryption programs and indirectly as the target language for verified high-level language compilers.

1.4. Achievements

Among the significant achievements of the Piton project are the following.

Piton truly provides abstract objects and a new programming language on top of a much lower level machine. Much has been written about this classic problem but the previous attempts to deal with it formally and mechanically have been unsatisfactory. We have in mind specifically the work related to the SRI Hierarchical Design Methodology [16] and its use in the Provably Secure Operating System (PSOS) [14] and the Software Implemented Fault Tolerant (SIFT) operating system [12, 19]. While virtually all of the issues are correctly intuited, we personally find great joy in seeing their formalization.

The commitment to stacking has had several effects. The desire to implement Piton forced into its design such practical considerations as the finite resources of the host machine. The desire to use Piton forced us to reflect the resource limits into the language itself—programs that cannot anticipate the imminent exhaustion of their resources are impractical.

We obtain efficiency in the Piton implementation by exploiting the fact that Piton programs are to be proved correct. In particular, the Piton semantics identifies ''erroneous'' computations and the Piton compiler is proved correct only for non-erroneous computations. By assuming the computation to be non-erroneous the compiler can generate more efficient code. But the only way to establish that a given computation is non-erroneous is to prove it from the formal semantics.

Developing the statement of the correctness result for the Piton implementation was very illuminating. Initially the idea was that the final state produced by a Piton computation could be alternatively produced by mapping the initial Piton state down to the FM8502, running the FM8502 to obtain a final state, and then mapping it up. The problem with this, aside from the issue of erroneous computations, is that we cannot map FM8502 states up to Piton states because not enough information is present. For example, we do not know how to interpret the FM8502 bit vectors in the data segment of the final FM8502 state: Are they natural numbers? Integers? Addresses? Because Piton syntax is untyped, it is impossible to determine the type of the result of a Piton program from a syntactic analysis of the program. Yet untyped languages are useful. They can be used because the user *knows* what the type of the result *will be* and interprets the final bit vectors accordingly. Our correctness result formalizes this notion of ‘‘foreknowledge.’’

The proof of the correctness result was, of course, the hardest task and the place where we learned the most. The implementation involves several phases: compilation, assembling, and linking.[1] Each phase is essentially a form of translation from one programming language to another. The topmost language is Piton, whose formal definition is part of the statement of the problem. The bottom-most language is FM8502 machine code, whose formal definition is embodied in an imagined physical machine and which is the *only* language that is physically implemented. But the compiler produces assembly code and the assembler produces symbolically linked machine code. One need not know about, much less formally define, these intermediate languages to *state* the correctness result. One need not know about, much less formally define, these languages to *use* the FM8502 implementation of Piton. It is even possible to *implement* Piton on FM8502 without ever writing down the definitions of these languages. And yet to *prove* the implementation correct we had to define these intermediate languages formally. In this sense, the Piton machine is three layers above FM8502, each layer implementing a Piton abstraction on top of a more primitive machine. Perhaps the most useful part of the Piton experience was recognizing the need for these layers and learning what kinds of problems could best be handled at each layer.

1.5. Outline of the Presentation

In Chapter 2 we informally describe the Piton programming language. We essentially adopt the style of a conventional primer for a programming language. We discuss such basic design issues as procedure call, errors, the various resources available, etc. We exhibit many examples. We summarily describe each instruction. The material in Chapter 2 is spiritually correct but, in the manner of most programming language primers, most of it is incomplete or technically incorrect.[2]

In Chapter 3 we illustrate Piton and the ideas discussed in Chapter 2 with a thoroughly worked example. In particular, we deal with the problem of ‘‘big number addition.’’ We explain (both informally and formally) what ‘‘big numbers’’ are, how to ‘‘add’’ them, and what the relation is between addition and big number addition. We then exhibit a Piton program that purportedly does big number addition. We exhibit a Piton initial state in which a particular big number addition computation is set up and we show the state obtained by running that initial state. We then exhibit the formal specification of the Piton program, we comment on the utility of our style of specification, and we discuss the mechanically checked proof that the program satisfies its specification. We return to this example when we discuss the implementation of Piton on FM8502 and the correctness theorem for the implementation.

In Chapter 4 we briefly sketch FM8502.

In Chapter 5 we state the correctness theorem for the FM8502 implementation of Piton, we informally characterize the various predicates and functions used in the theorem, and we explain the intended interpretation of the theorem. We then illustrate how the correctness theorem can be applied to the big number addition program developed in Chapter 3.

---

[1]Our implementation does involve three distinct phases, however the assembler and the linker are intertwined in the last phase. The first phase is what we call ‘‘resource representation’’ in which FM8502 resources are allocated to represent ‘‘system data’’ implicit in the Piton machine. The second phase is compilation. The third is what we call ‘‘link-assembling.’’ These distinctions are unimportant to the current discussion but are presented in detail later.

[2]We advocate formal definition of programming languages, but we recognize the pedagogical importance of informal ones.

In Chapter 6 we explain how Piton is implemented on FM8502. We give an example of an FM8502 core image produced from a Piton state, we explain the basic use of the FM8502 resources in our implementation, and we then discuss each of the three phases of the implementation: resource allocation, compilation, and link-assembling.

The next four chapters, Chapters 7-10, are the technical heart of this report. Chapter 7 contains the equations that define Piton. Chapter 8 contains the equations that define the machine language of FM8502. Chapter 9 contains the equations that define the implementation of Piton on FM8502. Chapter 10 contains the statement of the correctness result and the definitions of all of the concepts used in that theorem (except those contained in the foregoing chapters). All four of these chapters begin with brief, informal ‘‘guided tours’’ through the systems defined.

Finally, in Chapter 11 we briefly discuss the proof of the correctness theorem. Our primary motivation in this report is to convey accurately what has been proved—i.e., to explain Piton, FM8502, the implementation, and the correctness theorem—rather than how it was proved. Readers interested in the mechanical proof should contact the author after thoroughly digesting this report and its description of the proof.

There are two appendices. The first contains a complete list of all of the primitive function symbols occurring in Chapters 7-10. The second contains some statistics about the Piton project.

This report is exhaustively indexed. Approximately 600 function names are defined in Chapters 7-10. The index indicates the page number on which each function symbol is defined and lists all of the page numbers in Chapters 7-10 on which each function is used.

1.6. Notation

This report assumes a basic familiarity with the Boyer-Moore logic [4]. In examples of Piton syntax, states and data—all of which are explicit values in the logic—we frequently omit the quotation mark required to distinguish constants from formulas. For example, we sometimes refer to ‘‘the array’’ **`(3 2 1 0 0)`** rather than the **`LISTP`** object **`'(3 2 1 0 0)`**, or the ‘‘Piton instruction’’ **`(PUSH-LOCAL A)`** rather than the **`LISTP`** object **`'(PUSH-LOCAL A)`**. This sloppiness is intended to make the reader who is unfamiliar with our notation less burdened by details. It is hoped that the reader who is familiar with the notation will read these constants as though they were quoted and refer to the formal chapters of this report if ambiguity has accidentally crept in. When terms other than constants are presented we have adhered to our usual discipline of writing them in the implemented syntax of our logic.

1.7. Acknowledgements

The Piton project benefited substantially from the contributions of Warren Hunt, Matt Kaufmann, and Bill Young. Warren showed me how to program in FM8502 machine code, helped write the first version of the linker, and produced FM8502 from FM8501 in response to my requests. Matt volunteered to help construct the correctness proof and ‘‘contracted’’ to deliver the proof for one of the three main lemmas. I can think of no higher testimony to his mathematical and clerical skills than merely to point out that he was given a formula involving, at some level, about 500 defined function symbols and two months later delivered his proof—after finding and correcting dozens of bugs. His participation in the proof effort sped the whole project up far more than suggested by his two months of work. Finally, Bill is the first user of Piton—it is the target language of his MicroGypsy compiler—and so he has had the burden of being the ‘‘bad guy’’ who always needed some (perfectly reasonable) feature I had omitted. Without him, Piton would be far more of a toy than it is. The historical discussion of the Piton project in Appendix II more precisely describes the technical contributions of these three people. I would also like to thank Matt Wilding for his careful reading and constructive criticism of the first draft of this report.

Finally, I thank my wife, Miren, for her love and devotion during a long hard year.

2. An Informal Sketch of Piton

Among the features provided by Piton are:

- execute-only program space
- named read/write global data spaces randomly accessed as one-dimensional arrays
- recursive subroutine call and return
- provision of named formal parameters and stack-based parameter passing
- provision of named temporary variables allocated and initialized to constants on call
- a user-visible temporary stack
- seven abstract data types:
    - integers
    - natural numbers
    - bit vectors
    - Booleans
    - data addresses
    - program addresses (labels)
    - subroutine names
- stack-based instructions for manipulating the various abstract objects
- standard flow-of-control instructions
- instructions for determining resource limitations

2.1. An Example Piton Program

We begin our presentation of Piton with a simple example. Below we exhibit a Piton program named **DEMO**. The program is a list constant in the computational logic of Boyer and Moore [4] and is displayed in the traditional Lisp-like notation. Comments are written in the right-hand column, bracketed by the comment delimiters semi-colon and end-of-line. The **DEMO** program has three formal parameters, **X**, **Y**, and **Z**, and two temporary variables, **A** and **I**.

```
(DEMO (X Y Z)                          ; Formals X, Y, and Z
      ((A (INT -1))                    ; Temporary A, initial value -1
       (I (NAT 2)))                    ; Temporary I, initial value 2
      (PUSH-LOCAL Y)                   ; Push the value of Y
      (PUSH-CONSTANT (NAT 4))          ; Push the natural number 4
      (ADD-NAT)                        ; Add the top two items
      (RET))                           ; Return
```

When **DEMO** is called, the topmost three items from Piton's temporary stack are popped off and used as the actual values of the formals **X**, **Y**, and **Z**. In addition, **A** is initialized to the integer -1 and **I** is initialized to the natural number 2. The values of all five of these ‘‘local’’ variables are restored when **DEMO** returns to its caller.

The body of **DEMO** has four Piton instructions in it. The first, **(PUSH-LOCAL Y)**, pushes the value of the local variable **Y** onto the temporary stack. The second, **(PUSH-CONSTANT (NAT 4))**, pushes the natural number 4 onto the temporary stack. The third, **(ADD-NAT)**, pops the topmost two items off the temporary stack, adds them together (expecting both to be naturals), and pushes the result onto the temporary stack. The last instruction returns control to the calling environment. The sum just computed is on top of the stack and is considered the result. In summary, this silly program adds 4 to the value of its second argument and ignores the other arguments. Its two temporary variables are not used.

Now consider the following sequence of Piton instructions.

```
(PUSH-CONSTANT (ADDR (DELTA1 . 25)))
(PUSH-CONSTANT (NAT 17))
(PUSH-CONSTANT (BOOL T))
(CALL DEMO)
```

This sequence pushes three items onto the stack and then calls **DEMO**. The **CALL** pops the three objects off the stack and uses them as the actuals. **DEMO**'s first formal, **X**, is bound to the data address **(DELTA1 . 25)**—the address of the 25th location of the global array named **DELTA1**. **DEMO**'s second argument, **Y**, is bound to the natural number 17. Its third argument, **Z**, is bound to the Boolean value **T**. The execution of **DEMO** pushes 21 (the sum of 17 and 4) and returns. Thus, the net effect of the four instructions above—barring a variety of runtime errors such as stack overflow—is to push a 21 onto the stack.

2.2. Piton States

The Piton machine is a conventional von Neumann state transition machine. Roughly speaking, a particular instruction is singled out as the ‘‘current instruction’’ in any Piton state. When ‘‘executed’’ each instruction changes the state in some way, including changing the identity of the current instruction. The Piton machine operates on an initial state by iteratively executing the current instruction until some termination condition is met.

A Piton state, or *p-state*, is a 9-tuple with the following components:

- a *program counter*, indicating which instruction in which subroutine is the next to be executed;
- a *control stack*, recording the hierarchy of subroutine invocations leading to the current state;
- a *temporary stack*, containing intermediate results as well as the arguments and results of subroutine calls;
- a *program segment*, defining a system of Piton programs or subroutines;
- a *data segment*, defining a collection of disjoint named indexed data spaces (i.e., global arrays);
- a *maximum control stack size*;
- a *maximum temporary stack size*;
- a *word size*, which governs the size of numeric constants and bit vectors; and
- a *program status word* (*psw*).

The formalization of this concept is embodied in the function **P-STATE** which is defined on page 131. **P-STATE** takes nine arguments and returns a p-state with the appropriate nine components. Thus, **(P-STATE PC CSTK TSTK PROGS DATA MAXC MAXT W PSW)** is a p-state.

We put a variety of additional restrictions on the components of a p-state. For example, we require that every instruction in every program is syntactically well-formed and mentions no variables other than the locals of the containing program or the globals declared in the data segment. We also require that every data object occurring in the state is compatible with the state, e.g., every object tagged ‘‘address’’ is a legal address in that state, etc. We call such p-states *proper p-states*. The formalization of this concept is embodied in the function **PROPER-P-STATEP** which is defined on page 149.

The program counter of a p-state names one of the programs in the program segment, which we call the *current program*, and gives the position of one of the instructions in that program’s body, which we call the *current instruction*. We say *control* is *in* the current program and *at* the current instruction.

The control stack of the p-state is a stack of *frames*, the topmost frame describing the currently active subroutine invocation and the successive frames describing the hierarchy of suspended invocations. The topmost frame is the only frame directly accessible to Piton instructions. Each frame has two fields in it. One contains the *bindings* of the local variables of the invoked program. The other contains the *return program counter*, which is the program counter to which control is to return when the subroutine exits.

When a subroutine is *called* or *invoked*, a new frame is pushed onto the control stack. The local variables of the called subroutine are bound to the appropriate values and the return program counter is saved. Then control is transferred to the first instruction in the body of the subroutine. All references to local variables in the instructions of the called subroutine refer implicitly to the current bindings. When the subroutine returns to its caller, the top frame of the control stack is popped off, thus restoring the current bindings of the caller extant at the time of call. In short, the values assigned to the local variables of a subroutine are local to a particular invocation and cannot be accessed or changed by any other subroutine or recursive invocation. We define ‘‘local variables’’ and what we mean by the ‘‘appropriate values’’ when we discuss Piton programs.

2.3. Type Checking

Piton programs manipulate seven types of data: integers, natural numbers, Booleans, fixed length bit vectors, data addresses, program addresses, and subroutine names.

All objects are ‘‘first class’’ in the sense that they can be passed around and stored into arbitrary variable, stack, and data locations. *There is no type checking in the Piton syntax.* A variable can hold an integer value now and a Boolean value later, for example.

Each type comes with a set of Piton instructions designed to manipulate objects of that type. For example, the **ADD-NAT** instruction adds two naturals together to produce a natural; the **ADD-ADDR** instruction increments a data address by a natural to produce a new data address. The effects of most instructions are defined only when the operands are of the expected type. For example, the formal definition of Piton does not specify what the **ADD-NAT** instruction does if given a non-natural. However, our implementation of Piton *has no runtime type checking facilities.* The programmer must know what he is doing.

Such cavalier runtime treatment of types—i.e., no syntactic type checking and no runtime type checking—would normally be an invitation to disaster. In most programming languages the definition of the language is embodied in only two mechanical devices: the compiler (where syntactic checks are made) and the runtime system (where semantic checks are made). If some feature of the language (e.g., correct use of the type system) is not checked by either of these two devices, then the programmer had better read the language manual and his program very carefully because he bears the entire responsibility.

But the Piton programmer is relieved of this burden by an unconventional third mechanical device. In addition to a compiler and a run-time system, Piton has a mechanized formal semantics. This device—actually the Boyer-Moore theorem prover initialized with the formal definition of Piton—completely embodies the formal semantics of Piton. If a programmer wishes to establish that he has not violated Piton's type restrictions, he can undertake to prove it mechanically.

As programmers we find this a marvelous state of affairs. We are relieved of the burden of syntactic restrictions in the language—objects can be slung around any way we please. We are relieved of the inefficiency of checking types at runtime. But we don't have to worry about having made mistakes. The price, of course, is that we must be willing to prove our programs correct.

2.4. Data Types

As noted, Piton supports seven primitive data types. The syntax of Piton requires that all data objects be tagged by their type. Thus, **`(INT 5)`** is the way we write the integer 5, while **`(NAT 5)`** is the way we write the natural number 5. The question ''are they the same?'' cannot arise in Piton because no operation compares them.

Below we characterize all of the legal instances of each type. However, this must be done with respect to a given p-state, since the p-state determines the resource limitations, legal addresses, etc. We use w as the word size of the p-state implicit in our discussion. In the examples of this section we assume the word size is 8.[3] The formalization of the concept of ''legal Piton data object'' is embodied in the function **`P-OBJECTP`** which is defined on page 122.

2.4.1. Integers

Piton provides the integers, i, in the range $-2^{w-1} \le i < 2^{w-1}$. We say such integers are *representable* in the given p-state. Observe that there is one more representable negative integer than representable positive integers. Integers are written down in the form **`(INT i)`**, where **`i`** is an optionally signed integer in decimal notation. For example, **`(INT -4)`** and **`(INT 3)`** are Piton integers. Piton provides instructions for adding, subtracting, and comparing integers. It is also possible to convert non-negative integers into naturals.

2.4.2. Natural Numbers

Piton provides the natural numbers, n, in the range $0 \le n < 2^{w}$. We say such naturals are *representable* in the given p-state. Naturals are written down in the form **`(NAT n)`**, where **`n`** is an unsigned integer in decimal notation. For example, **`(NAT 0)`** and **`(NAT 7)`** are Piton naturals. Piton provides instructions for adding, subtracting, doubling, halving, and comparing naturals. Naturals also play a role in those instructions that do address manipulation, random access into the temporary stack, and some control functions.

---

[3]In our FM8502 implementation of Piton we fix the word size at 32.

2.4.3. Booleans

There are two Boolean objects, called **`T`** and **`F`**. They are written down **`(BOOL T)`** and **`(BOOL F)`**.[4] Piton provides the logical operations of conjunction, disjunction, negation and equivalence. Several Piton instructions generate Boolean objects (e.g., the ‘‘less than’’ operators for integers and naturals).

2.4.4. Bit Vectors

A Piton bit vector is an array of 1’s and 0’s as long as the word size. Bit vectors are written in the form **`(BITV v)`** where v is a list of length w, enclosed in parentheses, containing only 1’s and 0’s. For example (BITV (1 1 1 1 0 0 0 0)) is a bit vector when w is 8. Operations on bit vectors include componentwise conjunction, disjunction, negation, exclusive-or, left and right shift, and equivalence.

2.4.5. Data Addresses

A Piton data address is a pair consisting of a name and a number. To be legal in a given p-state, the name must be the name of some data area in the data segment of the state and the number must be non-negative and less than the length of the array associated with the named data area. Data addresses are written **`(ADDR (name . n))`**. Such an address refers to the **`n`**th element of array associated with **`name`**, where enumeration is 0 based, starting at the left hand end of the array. For example, if the data segment of the state contains a data area named **`DELTA1`** that has an associated array of length 128, then **`(ADDR (DELTA1 . 122))`** is a data address. The operations on data addresses include incrementing, decrementing, and comparing addresses, fetching the object at an address, and depositing an object at an address.

2.4.6. Program Addresses

A Piton program address is a pair consisting of a name and a number. To be legal in a given p-state, the name must be the name of some program in the program segment of the state and the number must be non-negative and less than the length of the body of the named program. Program addresses are written **`(PC (name . n))`**. Such an address refers to the **`n`**th instruction in the body of the program named **`name`**, where enumeration is 0 based starting with the first instruction in the body. For example, if the program segment of the state contains a program named **`SETUP`** that has 200 instructions in its body, then **`(PC (SETUP . 27))`** is a legal program address. Program addresses can be compared and control can be transferred to (the instruction at) a program address. Some instructions generate program addresses. But it is impossible to deposit anything at a program address (just as it is impossible to transfer control to a data address).

The program counter component of a p-state is an object of this type. For example, to start a computation at the first instruction of the program named **`MAIN`**, the program counter in the state should be set to **`(PC (MAIN . 0))`**.

---

[4] Note to those familiar with our logic: The **`T`** and **`F`** used in the representation of the Piton Booleans are *not* the **`(TRUE)`** and **`(FALSE)`** of the logic but the literal atoms **`'T`** and **`'F`** of the logic.

2.4.7. Subroutines

A Piton subroutine name is just a name. To be legal, it must be the name of some program in the program segment. Subroutine names are written **(SUBR name)**. For example, if **SETUP** is the name of a program in the program segment, then **(SUBR SETUP)** is a subroutine object in Piton. The only operation on subroutine objects is to call them.

2.5. The Data Segment

The Piton data segment contains all of the global data in a p-state. The data segment is a list of *data areas*. Each data area consists of a literal atom *data area name* followed by one or more Piton objects, called the *array* associated with the name. The objects in the array are implicitly indexed from 0, starting with the leftmost. Using data addresses, which specify a name and an index, Piton programs can access and change the elements in an array.

We sometimes call a data area name a *global variable*. Some Piton instructions expect global variables as their arguments and operate on the 0^{th} position of the named data area. We define the *value* of a global variable to be the contents of the 0^{th} location in its associated array. This is a pleasant convention if the data area only has one element but tends to be confusing otherwise.

Here, for example, is a data segment:

```
((LEN (NAT 5))
 (A   (NAT 0)
      (NAT 1)
      (NAT 2)
      (NAT 3)
      (NAT 4))
 (X   (INT -23)
      (NAT 256)
      (BOOL T)
      (BITV (1 0 1 0 1 1 0 0))
      (ADDR (A . 3))
      (PC (SETUP . 25))
      (SUBR MAIN))).
```

This segment contains three data areas, **LEN**, **A**, and **X**. The **LEN** area has only one element and so is naturally thought of as a global variable. Its value is the natural number 5. The **A** array is of length 5 and contains the consecutive naturals starting from 0. While **A** is of homogeneous type as shown, Piton programs may write arbitrary objects into **A**. The third data area, **X**, has an associated array of length 7. It happens that this array contains one object of every Piton type.

Let **addr** be the Piton data address object **(ADDR (X . 1))**. If we fetch from **addr** we get **(NAT 256)**. If we deposit **(NAT 7)** at **addr** the data segment becomes

```
((LEN (NAT 5))
 (A   (NAT 0)
      (NAT 1)
      (NAT 2)
      (NAT 3)
      (NAT 4))
 (X   (INT -23)
      (NAT 7)
      (BOOL T)
      (BITV (1 0 1 0 1 1 0 0))
      (ADDR (A . 3))
      (PC (SETUP . 25))
      (SUBR MAIN))).
```

If we increment **addr** by one and then fetch from **addr** we get **(BOOL T)**.

The individual data areas are totally isolated from each other. Despite the fact that addresses can be incremented and decremented, there is no way for a Piton program to manipulate **addr**, which addresses the area named **X**, so as to obtain an address into the area named **A**.

2.6. The Program Segment

The program segment of a p-state is a list of ''program definitions''. A *program definition* (or, interchangeably, a *subroutine definition*) is an object of the following form

$$\begin{array}{l}\texttt{(name (}v_0\ v_1\ \ldots\ v_{n-1}\texttt{)}\\ \quad\texttt{((}v_n\ i_n\texttt{) (}v_{n+1}\ i_{n+1}\texttt{)}\ \ldots\ \texttt{(}v_{n+k-1}\ i_{n+k-1}\texttt{))}\\ \quad ins_0\\ \quad ins_1\\ \quad \ldots\\ \quad ins_m\texttt{),}\end{array}$$

where **name** is the *name* of the program and is some literal atom; v_0, ..., v_{n-1} are the $n \geq 0$ *formal parameters* of the program and are literal atoms; v_n, ..., v_{n+k-1} are the $k \geq 0$ *temporary variables* of the program and are literal atoms; i_n, ..., i_{n+k-1} are the *initial values* of the corresponding temporary variables and are data objects in the state in which the program occurs; and ins_0, ... ins_m are $m+1$ optionally labelled Piton instructions, called the *body* of the program. The body must be non-empty.

The *local variables* of a program are the formal parameters together with the temporary variables. The values of the local variables of a subroutine may be accessed and changed by position as well as by name. For this purpose we enumerate the local variables starting from 0 in the same order they are displayed above.

As noted previously, upon subroutine call the local variables of the called subroutine are bound to the ''appropriate values'' in the stack frame created for that invocation. The **n** formal parameters are initialized from the temporary stack. The topmost **n** elements of the temporary stacks are called the *actuals* for the call. They are removed from the temporary stack and become the values of formals. The association is in reverse order of the formals; i.e., the last formal, v_{n-1}, is bound to the object on the top of the temporary stack at the time of the call and v_0 is bound to the object **n** down from the top at the time of the call. The **k** temporary variables are bound to their respective initial values at the time of call.

The instructions in the body of a Piton program may be optionally labelled. A *label* is a literal atom. To attach label **lab** to an instruction, **ins**, write

```
(DL lab comment ins)
```

where **comment** is any object in the logic and is totally ignored by the Piton semantics and implementation. We say **lab** is *defined* in a program if the body of the program contains **(DL lab ...)** as one of its members. Such a form is called a *def-label form* because it defines a label. Label definitions are local to the program in which they occur. Use of the atom **LOOP**, for example, as a label in some instruction in a program refers to the (first) point in that program at which **LOOP** is defined in a def-label form.

Because of the local nature of label definitions it is not possible for one program to jump to a label in another. A similar effect can be obtained efficiently using the data objects of type **PC**. The **POPJ** instruction transfers control to the program address on the top of the temporary stack—provided that address is in the current program.[5] Thus, if subroutine **MASTER** wants to jump to the 22nd instruction of subroutine **SLAVE**, it could **CALL SLAVE** and pass it the argument **(PC (SLAVE . 22))**, and **SLAVE** could do a **POPJ** as its first instruction to branch to the desired location.

The last instruction, $\mathbf{ins}_m$ must be a return or some form of unconditional jump. It is not permitted to ‘‘fall off’’ the end of a Piton program.

The formalization of the concept of a syntactically well-formed Piton program is embodied in the function **PROPER-P-PROGRAMP** which is defined on page 147. Part of the constraints on proper p-states is that they contain proper Piton programs.

2.7. Instructions

Figure 2-1 lists the Piton instructions. The instructions are organized informally into groups. The language as currently defined provides 65 instructions. We do not regard the current instruction set as fixed in granite; we imagine Piton will continue to evolve to suit the needs of its users.

We describe each instruction informally by explaining the syntactic form of the instruction, the preconditions on its execution, and the effects of executing an acceptable instance of the instruction. The instructions are listed in alphabetical order. Unless otherwise indicated, every instruction increments the program counter by one so that the next instruction to be executed is the instruction following the current one in the current subroutine. All references to ‘‘the stack’’ refer to the temporary stack unless otherwise specified. When we say ‘‘push’’ or ‘‘pop’’ without mentioning a particular stack we mean to push or pop the temporary stack. The formal section of this document is intended as a readable and precise Piton manual. The references to page numbers below refer to the formal definitions of the corresponding functions or predicates. The first section of Chapter 7 is a guide to the formalization.

(ADD-ADDR) *Well Formedness* (page 140): No additional constraints. *Precondition* (page 102): There is a natural, n, on top of the stack and a data address, a, immediately below it. The result of incrementing a by n is a legal data address. *Effect* (page 102): Pop twice and then push the data address obtained by incrementing a by n.

(ADD-INT) *Well Formedness* (page 140): No additional constraints. *Precondition* (page 102): There is an integer, i, on top of the stack and an integer, j, immediately below it. j+i is representable. *Effect* (page 103): Pop twice and then push the integer j+i.

[5] There is no way, in Piton, to transfer control into another subroutine except via the call/return mechanism.

Figure 2-1: Piton Instructions

Control

CALL
JUMP
JUMP-CASE
NO-OP
RET
TEST-BITV-AND-JUMP
TEST-BOOL-AND-JUMP
TEST-INT-AND-JUMP
TEST-NAT-AND-JUMP

Integers

ADD-INT
ADD-INT-WITH-CARRY
ADD1-INT
EQ
INT-TO-NAT
LT-INT
NEG-INT
SUB-INT
SUB-INT-WITH-CARRY
SUB1-INT

Natural Numbers

ADD-NAT
ADD-NAT-WITH-CARRY
ADD1-NAT
DIV2-NAT
EQ
LT-NAT
MULT2-NAT
MULT2-NAT-WITH-CARRY-OUT
SUB-NAT
SUB-NAT-WITH-CARRY
SUB1-NAT

Variables

LOCN
POP-GLOBAL
POP-LOCAL
POP-LOCN
PUSH-GLOBAL
PUSH-LOCAL
SET-GLOBAL
SET-LOCAL

Booleans

AND-BOOL
EQ
NOT-BOOL
OR-BOOL

Bit Vectors

AND-BITV
EQ
LSH-BITV
NOT-BITV
OR-BITV
RSH-BITV
XOR-BITV

Stack

DEPOSIT-TEMP-STK
FETCH-TEMP-STK
POP
POP*
POPN
PUSH-CONSTANT
PUSH-TEMP-STK-INDEX

Resources

JUMP-IF-TEMP-STK-EMPTY
JUMP-IF-TEMP-STK-FULL
PUSH-CTRL-STK-FREE-SIZE
PUSH-TEMP-STK-FREE-SIZE

Data Addresses

ADD-ADDR
DEPOSIT
EQ
FETCH
LT-ADDR
SUB-ADDR

Subroutines

EQ
POP-CALL

Program Addresses

EQ
POPJ
PUSHJ

(ADD-INT-WITH-CARRY)

Well Formedness (page 140): No additional constraints. *Precondition* (page 103): There is an integer, i, on top of the stack, an integer, j, immediately below it, and a Boolean, c, below that. *Effect* (page 103): Pop three times. Let k be 1 if c is **T** and 0 otherwise. Let sum be i+j+k. If sum is representable in the word size, w, of this p-state, push the Boolean **F** and then the integer sum; if sum is not representable and is negative, push the Boolean **T** and the integer sum+2^w; if sum is not representable and positive, push the Boolean **T** and the integer sum-2^w.

(ADD-NAT) *Well Formedness* (page 140): No additional constraints. *Precondition* (page 104):There is a natural, i, on top of the stack and a natural, j, immediately below it. j+i is representable. *Effect* (page 104): Pop twice and then push the natural j+i.

(ADD-NAT-WITH-CARRY)
Well Formedness (page 140): No additional constraints. *Precondition* (page 104): There is a natural, i, on top of the stack, a natural, j, immediately below it, and a Boolean, c, immediately below that. *Effect* (page 105): Pop three times. Let k be 1 if c is **T** and 0 otherwise. Let sum be the natural i+j+k. If sum is representable in the word size, w, of this p-state, push the Boolean **F** and then natural sum; if sum is not representable, push the Boolean **T** and the natural sum-2^w.

(ADD1-INT) *Well Formedness* (page 141): No additional constraints. *Precondition* (page 105): There is an integer, i, on top of the stack and i+1 is representable. *Effect* (page 105): Pop once and then push the integer i+1.

(ADD1-NAT) *Well Formedness* (page 141): No additional constraints. *Precondition* (page 106): There is a natural, i, on top of the stack and i+1 is representable. *Effect* (page 106): Pop once and then push the natural i+1.

(AND-BITV) *Well Formedness* (page 141): No additional constraints. *Precondition* (page 106): There is a bit vector, v1, on top of the stack and a bit vector, v2, immediately below it. *Effect* (page 106): Pop twice and then push the bit vector result of the componentwise conjunction of v1 and v2.

(AND-BOOL) *Well Formedness* (page 141): No additional constraints. *Precondition* (page 107): There is a Boolean, b1, on top of the stack and a Boolean, b2, immediately below it. *Effect* (page 107): Pop twice and then push the Boolean conjunction of b1 and b2.

(CALL subr) *Well Formedness* (page 141): **subr** is the name of a program in the program segment. *Precondition* (page 107): Suppose that **subr** has n formal variables and k temporary variables. Then the temporary stack must contain at least n items and the control stack must have at least 2+n+k free slots. *Effect* (page 108): Transfer control to the first instruction in the body of **subr** after removing the topmost n elements from the temporary stack and constructing a new frame on the control stack. In the new frame the formals of **subr** are bound to the n elements removed from the temporary stack, in reverse order, the temporaries of **subr** are bound to their declared initial values, and the return program counter points to the instruction after the **CALL**.

(DEPOSIT) *Well Formedness* (page 142): No additional constraints. *Precondition* (page 108): There is a data address, a, on top of the stack and an arbitrary object, val, immediately below it. *Effect* (page 109): Pop twice and then deposit val into the location addressed by a.

(DEPOSIT-TEMP-STK)
Well Formedness (page 142): No additional constraints. *Precondition* (page 109): There is a natural number, n, on top of the stack and some object, val, immediately below it. Furthermore, n is less than the length of the stack after popping two elements. *Effect* (page 109): Pop twice and then deposit val at the n^{th} position in the temporary stack, where positions are enumerated from 0 starting at the bottom.

(DIV2-NAT) *Well Formedness* (page 142): No additional constraints. *Precondition* (page 109): There is a natural, i, on top of the stack and room to push at least one more item. *Effect* (page 110): Pop once and then push the natural floor of the quotient of i divided by 2 and then push the natural i mod 2.

(EQ) *Well Formedness* (page 142): No additional constraints. *Precondition* (page 110): The temporary stack contains at least two items and the top two are of the same type. *Effect* (page 110): Pop twice and then push the Boolean **T** if they are the same and the Boolean **F** if they are not.

(FETCH) *Well Formedness* (page 142): No additional constraints. *Precondition* (page 110): There is a data address, a, on top of the stack. *Effect* (page 111): Pop once and then push the contents of address a.

(FETCH-TEMP-STK)
Well Formedness (page 142): No additional constraints. *Precondition* (page 111): There is a natural number, n, on top of the stack and n is less than the length of the stack. *Effect* (page 111): Let val be the n^{th} element of the stack, where elements are enumerated from 0 starting at the bottom-most element. Pop once and then push val.

(INT-TO-NAT) *Well Formedness* (page 144): No additional constraints. *Precondition* (page 114): There is a non-negative integer, i, on top of the stack. *Effect* (page 115): Pop and then push the natural i.

(JUMP lab) *Well Formedness* (page 145): **lab** is a label in the containing program. *Precondition* (page 116): None. *Effect* (page 116): Jump to **lab**.

(JUMP-CASE lab0 lab1 ... labn)
Well Formedness (page 144): Each of the **labi** is a label in the containing program. *Precondition* (page 115): There is a natural, i, on top of the stack and i ≤ **n**. *Effect* (page 115): Pop once and then jump to **lab**i.

(JUMP-IF-TEMP-STK-EMPTY lab)
Well Formedness (page 145): **lab** is a label in the containing program. *Precondition* (page 115): None. *Effect* (page 115): Jump to **lab** if the temporary stack is empty.

(JUMP-IF-TEMP-STK-FULL lab)
Well Formedness (page 145): **lab** is a label in the containing program. *Precondition* (page 116): None. *Effect* (page 116): Jump to **lab** if the temporary stack is full.

(LOCN lvar) *Well Formedness* (page 145): **lvar** is a local variable of the containing program. *Precondition* (page 116): The value, n, of **lvar** is a natural number less than the number of locals of the current program. *Effect* (page 117): Push the value of the n^{th} local variable of the current program.

(LSH-BITV) *Well Formedness* (page 145): No additional constraints. *Precondition* (page 117): There is a bit vector, v, on top of the stack. *Effect* (page 117): Pop once and then push the bit vector result of left shifting each bit of v, bringing in a 0 on the right.

(LT-ADDR) *Well Formedness* (page 145): No additional constraints. *Precondition* (page 117): There is a data address, a1, on top of the stack and a data address, a2, immediately below it. a1 and a2 address the same data area. *Effect* (page 118): Pop twice and then push the Boolean **T** if a2 < a1 (i.e., the position addressed by a2 is to the left of that addressed by a1) and push the Boolean **F** otherwise.

(LT-INT) *Well Formedness* (page 145): No additional constraints. *Precondition* (page 118): There is an integer, i, on top of the stack and an integer, j, immediately below it. *Effect* (page 118): Pop twice and then push the Boolean **T** if j < i and the Boolean **F** otherwise.

(LT-NAT) *Well Formedness* (page 145): No additional constraints. *Precondition* (page 118): There is a natural, i, on top of the stack and a natural, j, immediately below it. *Effect* (page 119): Pop twice and then push the Boolean **T** if j < i and the Boolean **F** otherwise.

(MULT2-NAT) *Well Formedness* (page 145): No additional constraints. *Precondition* (page 119): There is a natural, i, on top of the stack and 2*i is representable. *Effect* (page 119): Pop once and then push the natural 2*i.

(MULT2-NAT-WITH-CARRY-OUT)
Well Formedness (page 145): No additional constraints. *Precondition* (page 119): There is a natural, i, on top of the stack and room to push at least one more item. *Effect* (page 120): Pop once. Then, if the natural 2*i is representable, push the Boolean **F** and then the natural 2*i. Otherwise, push the Boolean **T** and the natural $2*i-2^w$, where w is the word size of this p-state.

(NEG-INT) *Well Formedness* (page 146): No additional constraints. *Precondition* (page 120): There is an integer, i, on top of the stack and -i is representable. *Effect* (page 120): Pop once and then push the integer -i.

(NO-OP) *Well Formedness* (page 146): No additional constraints. *Precondition* (page 120): None. *Effect* (page 121): Do nothing; continue execution.

(NOT-BITV) *Well Formedness* (page 146): No additional constraints. *Precondition* (page 121): There is a bit vector, v, on top of the stack. *Effect* (page 121): Pop once and then push the bit vector result of the componentwise logical negation of v.

(NOT-BOOL) *Well Formedness* (page 146): No additional constraints. *Precondition* (page 121): There is a Boolean, b, on top of the stack. *Effect* (page 121): Pop once and then push the Boolean negation of b.

(OR-BITV) *Well Formedness* (page 146): No additional constraints. *Precondition* (page 122): There is a bit vector, v1, on top of the stack and a bit vector, v2, immediately below it. *Effect* (page 122): Pop twice and then push the bit vector result of the componentwise disjunction of v1 and v2.

(OR-BOOL) *Well Formedness* (page 146): No additional constraints. *Precondition* (page 123): There is a Boolean, b1, on top of the stack and a Boolean, b2, immediately below it. *Effect* (page 123): Pop twice and then push the Boolean disjunction of b1 and b2.

(POP) *Well Formedness* (page 146): No additional constraints. *Precondition* (page 125): There is at least one item on the stack. *Effect* (page 126): Pop and discard the top of the stack.

(POP* n) *Well Formedness* (page 146): **n** is a natural number. Note: **(POP* 3)** is well formed; **(POP* (NAT 3))** is not. *Precondition* (page 123): there are at least **n** items on the stack. *Effect* (page 123): Pop and discard the topmost **n** items.

(POP-CALL) *Well Formedness* (page 146): No additional constraints. *Precondition* (page 124): A subroutine name, subr, is on top of the stack and, after removing that name, it is legal to **CALL** subr (i.e., sufficient arguments are on the temporary stack and the control stack has room for the new frame). *Effect* (page 124): Pop once and then execute **(CALL** subr**)**.

(POP-GLOBAL gvar)

Well Formedness (page 146): **gvar** is a global variable, i.e., the name of a data area in the data segment of the containing p-state. *Precondition* (page 124): There is an object, val, on top of the stack. *Effect* (page 124): Pop and assign val to the (0^{th} position of the array associated with the) global variable **gvar**.

(POP-LOCAL lvar)

Well Formedness (page 146): **lvar** is a local variable of the containing program. *Precondition* (page 124): There is an object, val, on top of the stack. *Effect* (page 125): Pop and assign val to the local variable **lvar**.

(POP-LOCN lvar)

Well Formedness (page 147): **lvar** is a local variable of the containing program. *Precondition* (page 125): The value, n, of **lvar** is less than the number of local variables of the current program and there is an object, val, on top of the stack. *Effect* (page 125): Pop and assign val to the n^{th} local variable.

(POPJ) *Well Formedness* (page 147): No additional constraints. *Precondition* (page 126): There is a program counter object, pc, on top of the stack and pc addresses the current program. *Effect* (page 126): Pop once and then transfer control to pc.

(POPN) *Well Formedness* (page 147): No additional constraints. *Precondition* (page 126): There is a natural, n, on top of the stack and there are at least n items on the stack below it. *Effect* (page 126): Pop and discard n+1 items. Thus, to pop n items off the stack, push n onto the stack and execute **(POPN)**.

(PUSH-CONSTANT const)

Well Formedness (page 147): **const** is either a legal Piton object in the containing p-state, the atom **PC**, or a label in the containing program. *Precondition* (page 127): There is room to push at least one item. *Effect* (page 127): If **const** is a Piton object, push **const**; if **const** is the atom **PC**, push the program counter of the next instruction; otherwise push the program counter corresponding to the label **const**.

(PUSH-CTRL-STK-FREE-SIZE)

Well Formedness (page 147): No additional constraints. *Precondition* (page 127): There is room to push at least one item. *Effect* (page 127): Push the natural number indicating how many more cells can be created on the control stack before the maximum control stack size is exceeded.

(PUSH-GLOBAL gvar)

Well Formedness (page 148): **gvar** is a global variable, i.e., the name of a data area in the data segment of the containing p-state. *Precondition* (page 127): There is room to push at least one item. *Effect* (page 128): Push the value of the global variable **gvar**, i.e., the contents of position 0 in the array associated with **gvar**.

(PUSH-LOCAL lvar)

Well Formedness (page 148): **lvar** is a local variable of the containing program. *Precondition* (page 128): There is room to push at least one item. *Effect* (page 128): Push the value of the local variable **lvar**.

(PUSH-TEMP-STK-FREE-SIZE)

Well Formedness (page 148): No additional constraints. *Precondition* (page 128): There is room to push at least one item. *Effect* (page 128): Push the natural number indicating how many more cells can be created on the temporary stack before the maximum temporary stack size is exceeded.

(PUSH-TEMP-STK-INDEX n)
Well Formedness (page 148): **n** is a natural number. Note: **n** here must not be tagged; **(PUSH-TEMP-STK-INDEX 3)** is well formed and **(PUSH-TEMP-STK-INDEX (NAT 3))** is not. *Precondition* (page 129): **n** is less than the length of the temporary stack and there is room to push at least one item. *Effect* (page 129): Push the natural number (length-**n**)-1, where length is the current length of the temporary stack. Note: We permit the temporary stack to be accessed randomly as an array. The elements in the stack are enumerated from 0 starting at the *bottom-most* so that pushes and pops do not change the positions of undisturbed elements. This instruction converts from a topmost-first enumeration to our enumeration. That is, it pushes onto the temporary stack the index of the element **n** removed from the top. See also **FETCH-TEMP-STK** and **DEPOSIT-TEMP-STK**.

(PUSHJ lab) *Well Formedness* (page 148): **lab** is a label in the containing program. *Precondition* (page 129): There is room to push at least one item. *Effect* (page 129): Push the program counter addressing the next instruction and then jump to **lab**.

(RET) *Well Formedness* (page 148): No additional constraints. *Precondition* (page 129): None. *Effect* (page 130): If the control stack contains only one frame (i.e., if the current invocation is the top-level entry into Piton) **HALT** the machine. Otherwise, set the program counter to the return program counter in the topmost frame of the control stack and pop that frame off the control stack.

(RSH-BITV) *Well Formedness* (page 148): No additional constraints. *Precondition* (page 130): There is a bit vector, v, on top of the stack. *Effect* (page 130): Pop once and then push the bit vector result of right shifting each bit in v, bringing in a 0 on the left.

(SET-GLOBAL gvar)
Well Formedness (page 148): **gvar** is a global variable, i.e., the name of a data area in the data segment of the containing p-state. *Precondition* (page 130): There is an object, val, on top of the stack. *Effect* (page 130): Assign val to the (0^{th} position of the array associated with the) global variable **gvar**. The stack is not popped.

(SET-LOCAL lvar)
Well Formedness (page 148): **lvar** is a local variable in the containing program. *Precondition* (page 131): There is an object, val, on top of the stack. *Effect* (page 131): Assign val to the local variable **lvar**. The stack is not popped.

(SUB-ADDR) *Well Formedness* (page 149): No additional constraints. *Precondition* (page 131): There is a natural, n, on top of the stack and a data address, a, immediately below it. The result of decrementing a by n is a legal data address. *Effect* (page 132): Pop twice and then push the data address obtained by decrementing a by n.

(SUB-INT) *Well Formedness* (page 149): No additional constraints. *Precondition* (page 132): There is an integer, i, on top of the stack and an integer, j, immediately below it. j-i is representable. *Effect* (page 132): Pop twice and then push the integer j-i.

(SUB-INT-WITH-CARRY)
Well Formedness (page 149): No additional constraints. *Precondition* (page 133): There is an integer, i, on top of the stack, an integer, j, immediately below it, and a Boolean, c, below that. *Effect* (page 133): Pop three times. Let k be 1 if c is **T** and 0 otherwise. Let diff be the integer j-(i+k). If diff is representable in the word size, w, of this p-state, push the Boolean **F** and the integer diff; if diff is not representable and is negative, push the Boolean **T** and the integer diff+2^w; if diff is not representable and is positive, push the Boolean **T** and the integer diff-2^w.

(SUB-NAT) *Well Formedness* (page 149): No additional constraints. *Precondition* (page 134): There is a natural, i, on top of the stack and natural, j, immediately below it. Furthermore, j ≥ i. *Effect* (page 134): Pop twice and then push the natural j-i.

(SUB-NAT-WITH-CARRY)
Well Formedness (page 149): No additional constraints. *Precondition* (page 134): There is a natural, i, on top of the stack, a natural, j, immediately below it, and a Boolean, c, immediately below that. *Effect* (page 135): Pop three times. Let k be 1 if c is **T** and 0 otherwise. If j ≥ i+k, then push the Boolean **F** and the natural j-(i+k). Otherwise, push the Boolean **T** and the natural 2^w-((i+k)-j), where w is the word size of this p-state.

(SUB1-INT) *Well Formedness* (page 149): No additional constraints. *Precondition* (page 135): There is an integer, i, on top of the stack and i-1 is representable. *Effect* (page 136): Pop once and then push the integer i-1.

(SUB1-NAT) *Well Formedness* (page 150): No additional constraints. *Precondition* (page 136): There is a non-zero natural, i, on top of the stack. *Effect* (page 136): Pop and then push the natural i-1.

(TEST-BITV-AND-JUMP test lab)
Well Formedness (page 150): **test** is either **ALL-ZERO** or **NOT-ALL-ZERO** and **lab** is a label in the containing program. *Precondition* (page 137): There is a bit vector, v, on top of the stack. *Effect* (page 137): Pop once and then jump to **lab** if **test** is satisfied, as indicated below.

test	condition tested
ALL-ZERO	every component of v is 0
NOT-ALL-ZERO	some component of v is 1

(TEST-BOOL-AND-JUMP test lab)
Well Formedness (page 150): **test** is either **T** or **F** and **lab** is a label in the containing program. *Precondition* (page 137): There is a Boolean, b, on top of the stack. *Effect* (page 138): Pop once and then jump to **lab** if **test** is satisfied, as indicated below.

test	condition tested
T	b = **T**
F	b = **F**

(TEST-INT-AND-JUMP test lab)
Well Formedness (page 150): **test** is one of **NEG**, **NOT-NEG**, **ZERO**, **NOT-ZERO**, **POS** or **NOT-POS**, and **lab** is a label in the containing program. *Precondition* (page 138): There is an integer, i, on top of the stack. *Effect* (page 138): Pop once and then jump to **lab** if **test** is satisfied, as indicated below.

test	condition tested
NEG	$i < 0$
NOT-NEG	$i \geq 0$
ZERO	$i = 0$
NOT-ZERO	$i \neq 0$
POS	$i > 0$
NOT-POS	$i \leq 0$

(TEST-NAT-AND-JUMP test lab)

Well Formedness (page 150): **test** is either **ZERO** or **NOT-ZERO** and **lab** is a label in the containing program.[6] *Precondition* (page 138): There is a natural, n, on top of the stack. *Effect* (page 138): Pop once and then jump to **lab** if **test** is satisfied, as indicated below.

test	condition tested
ZERO	$n = 0$
NOT-ZERO	$n \neq 0$

(XOR-BITV) *Well Formedness* (page 150): No additional constraints. *Precondition* (page 139): There is a bit vector, v1, on top of the stack and a bit vector, v2, immediately below it. *Effect* (page 139): Pop twice and then push the bit vector result of the componentwise exclusive-or of v1 and v2.

2.8. The Piton Interpreter

Associated with each instruction is a predicate on p-states called the *ok predicate* or the *precondition* for the instruction. This predicate insures that it is legal to execute the instruction in the current p-state. Generally speaking, the precondition of an instruction checks that the operands exist, have the appropriate types and do not cause the machine to exceed its resource limits.

Also associated with each instruction is a function from p-states to p-states called the *step* or *effects* function. The step function for an instruction defines the state produced by executing the instruction, provided the precondition is satisfied. Most of the step functions increment the program counter by one and manipulate the stacks and/or global data segment.

The Piton interpreter is a typical von Neumann state transition machine. The interpreter iteratively constructs the new current state by applying the step function for the current instruction to the current state, provided the precondition is satisfied. This process stops, if at all, either when a precondition is unsatisfied or a top-level return instruction is executed.[7] The property of being a proper p-state is preserved by the Piton interpreter. That is, if the initial state is proper, so is the final state. The formalization of the Piton interpreter is the function **P** which is defined on page 102. **(P s n)** is the p-state obtained by executing **n** instructions starting in p-state **s**.

2.9. Erroneous States

What does the Piton machine do when the precondition for the current instruction is not satisfied? This brings us to the role of the program status word, psw, in the state and its use in error handling. This has important consequences in the design, implementation, and proof of Piton.

The psw is normally set to the literal atom **RUN**, which indicates that the computation is proceeding normally. The psw is set to **HALT** by the **RET** (return) instruction when executed in the top level program; the **HALT** psw indicates successful termination of the computation. The psw is set to one of a many *error*

[6] Technically, **test** may be anything whatsoever. If it is not **ZERO** it is treated as though it were **NOT-ZERO**.

[7] We formalize this machine constructively by defining the function that iterates the process a given number of times.

conditions whenever the precondition for the current instruction is not satisfied. Any state with a psw other than **RUN** or **HALT** is called an *erroneous* state. The Piton interpreter is defined as an identity function on erroneous states.

No Piton instruction (i.e., no precondition or step function) inspects the psw. It is impossible for a Piton program to trap or mask an error. The psw and the notion of erroneous states are metatheoretic concepts in Piton; they are used to define the language but are not part of the language.

We consider an implementation of Piton correct if it has the property that it can successfully carry out every computation that produces a non-erroneous state. This is made formal when we present our correctness theorem. But the consequences to the implementation should be clear now. For example, the **ADD-NAT** instruction requires that two natural numbers be on top of the stack and that their sum is representable. This need not be checked at run-time by the implementation of Piton. The run-time code for **ADD-NAT** can simply add together the top two elements of the stack and increment the program counter. If the Piton machine produces a non-erroneous state on the **ADD-NAT** instruction, then the implementation follows it faithfully. If the Piton machine produces an erroneous state, then it does not matter what the implementation does. For example, our implementation of **ADD-NAT** does not check that the stack has two elements, that the top two elements are naturals, or that their sum is representable. It is difficult even to characterize the damage that might be caused if these conditions are not satisfied when our code is executed. As noted in our discussion of type checking, mechanical proof can be used to certify that no such errors occur.

The language contains adequate facilities to program explicit checks for all resource errors. For example, **ADD-NAT-WITH-CARRY** will not only add two naturals together, it will push a Boolean which indicates whether the result is the true sum. If you have to test whether the sum is representable, use **ADD-NAT-WITH-CARRY**. On the other hand, if you *know* the result is representable, use **ADD-NAT**.

But, unless you are adding constants together, how can you possibly know the result is representable? That is, under what conditions can you to use **ADD-NAT** and still prove the absence of errors? This brings us to the crux of the problem. When you write a Piton program and prove it non-erroneous you do not have to prove the total absence of errors. You *do* have to state the conditions under which the program may be called and prove the absence of errors under those conditions. For example, a typical hypothesis about the initial state might be that the sum of the top two elements of the stack is representable and the stack contains at least 5 free cells. These conditions are expressed in the logic, not in Piton. We illustrate the handling of errors in the next chapter.

Most of the rest of this report concerns the implementation of Piton on verified hardware and the proof of the correctness of the implementation. With three exceptions, this material is not relevant to the reader who simply wishes to use Piton. The three exceptions are Chapters 3, 5 and 7. Chapter 3 illustrates the use of Piton as a verifiable programming language: we specify, implement, and prove the correctness of a big number addition algorithm. Chapter 5 informally describes what was proved about the FM8502 implementation of Piton and can be regarded as the warranty that comes with the FM8502 Piton implementation. Chapter 7 gives the formal definition of Piton and thus serves as a precise reference manual for the language.

3. Big Number Addition

In this chapter we consider an example programming problem and its solution in Piton. The problem is to specify, implement, and verify a program for doing ‘‘big number addition.’’ This chapter is a rather long but instructive detour from our main goal of implementing Piton on FM8502 and proving the implementation correct.

3.1. An Informal Explanation of Big Number Addition

A ‘‘big number’’ is a fixed length array of ‘‘digits,’’ each digit being a natural number less than a fixed ‘‘base.’’ The intended interpretation of such an array is that it represents the natural number obtained by summing the product of the successive digits and successive powers of the base. In our representation of big numbers we put the least significant digit in position 0. For example, a big number array of length 5 representing the number 123 in base 10 is **`(3 2 1 0 0)`**. Of course, normally the base of a big number system is the first unrepresentable natural on the host machine. For example, in a 32-bit wide machine, the natural base for big number arithmetic is 2^{32}, so that each digit is a full word.

Big number addition is the process that takes as input two big number arrays and produces as output the big number array representing their sum. For example, the table below shows two naturals, their corresponding base 100 big-number arrays of length 5 and the two sums (the natural sum and the corresponding big number sum).

	12, 345, 678	(78 56 34 12 0)
+	70, 005, 020	(20 50 0 70 0)
	82, 350, 698	(98 6 35 82 0)

Given our representation of big numbers one adds corresponding digits starting at the leftmost, carrying to the right.

3.2. A Formal Explanation of Big Number Addition

A *big number (in base* **`base`***)* is an object **`a`** satisfying the predicate **`(BIGNP a base)`**, where

Definition.
```
(BIGNP A BASE)
   =
(IF (NLISTP A)
    (EQUAL A NIL)
    (AND (LISTP (CAR A))
         (EQUAL (TYPE (CAR A)) 'NAT)
         (NUMBERP (UNTAG (CAR A)))
         (LESSP (UNTAG (CAR A)) BASE)
         (EQUAL (CDDR (CAR A)) NIL)
         (BIGNP (CDR A) BASE)))).
```

The function **`TYPE`** is defined on page 153 as part of the formal definition of Piton. **`(TYPE x)`** returns the tag of the Piton object **`x`**. **`UNTAG`**, defined on page 153, strips off the tag of a Piton object. Thus, a big number is a proper list of tagged naturals, each of which is less than the base. An example big number in base 100 is **`'((NAT 78) (NAT 56) (NAT 34) (NAT 12) (NAT 0))`**.

The natural number *represented by* a big number **`a`** in base **`base`** is **`(BIGN->NAT a base)`**.

Definition.

```
(BIGN->NAT A BASE)
   =
(IF (NLISTP A)
    0
    (PLUS (UNTAG (CAR A))
          (TIMES BASE (BIGN->NAT (CDR A) BASE))))
```

For example, the natural represented by **'((NAT 78) (NAT 56) (NAT 34) (NAT 12) (NAT 0))** in base 100 is

$$78 + 100*(56 + 100*(34 + 100*(12 + 100*(0 + 0))))$$
$$=$$
$$78 + 56*100 + 34*100^2 + 12*100^3 + 0*100^4$$
$$=$$
$$78 + 5600 + 340000 + 12000000$$
$$=$$
$$12345678.$$

We define big number addition by the pair of functions shown below.

Definition.

```
(BIG-ADD-ARRAY A B C BASE)
   =
(IF (NLISTP A)
    NIL
    (CONS (TAG 'NAT (REMAINDER (PLUS (UNTAG (CAR A))
                                     (UNTAG (CAR B))
                                     (TV->NAT C))
                               BASE))
          (BIG-ADD-ARRAY (CDR A)
                         (CDR B)
                         (NOT (LESSP (PLUS (UNTAG (CAR A))
                                           (UNTAG (CAR B))
                                           (TV->NAT C))
                                     BASE))
                         BASE)))
```

Definition.

```
(BIG-ADD-CARRY-OUT A B C BASE)
   =
(IF (NLISTP A)
    (BOOL C)
    (BIG-ADD-CARRY-OUT (CDR A)
                       (CDR B)
                       (NOT (LESSP (PLUS (UNTAG (CAR A))
                                         (UNTAG (CAR B))
                                         (TV->NAT C))
                                   BASE))
                       BASE))
```

Both functions take as input two big numbers, **A** and **B**, an ''input carry flag'', **C**, and the specified base, **BASE**. We assume **A** and **B** are both of length n. The first function, **BIG-ADD-ARRAY**, produces a big number of length n. The second function, **BIG-ADD-CARRY-OUT**, produces a truth value, called the *carry out*, which indicates whether the sum is too big to be represented in n digits. The functions **TAG** and **BOOL** used above are defined on pages 153 and 97 as part of the formal definition of Piton. **(TAG 'NAT n)** produces the tagged object **'(NAT n)** and **(BOOL c)** produces a tagged Piton Boolean from a truth value, e.g., `(BOOL T)` is `'(BOOL T)`. The subsidiary function `TV->NAT` converts a truth value to a natural and is defined as `(TV->NAT C)` = `(IF C 1 0)`.

Recall the previously shown example of big number addition.

	12, 345, 678	(78 56 34 12 0)
+	70, 005, 020	(20 50 0 70 0)
	82, 350, 698	(98 6 35 82 0)

We display the formal version of this example in two parts, the five digit addition,

```
(BIG-ADD-ARRAY
     '((NAT 78) (NAT 56) (NAT 34) (NAT 12) (NAT 0))
     '((NAT 20) (NAT 50) (NAT  0) (NAT 70) (NAT 0))
     F 100)
=
     '((NAT 98) (NAT  6) (NAT 35) (NAT 82) (NAT 0)),
```

and the determination that there is no carry out,

```
(BIG-ADD-CARRY-OUT
     '((NAT 78) (NAT 56) (NAT 34) (NAT 12) (NAT 0))
     '((NAT 20) (NAT 50) (NAT  0) (NAT 70) (NAT 0))
     F 100)
=
     F.
```

We can package `BIG-ADD-ARRAY` and `BIG-ADD-CARRY-OUT` into a single function, here called `BIG-PLUS`, which takes two big numbers of length n and returns the big number sum of length n+1 obtained by concatenating to the `BIG-ADD-ARRAY` a single high order digit obtained from the carry out.

Definition.

```
(BIG-PLUS A B C BASE)
   =
(APPEND (BIG-ADD-ARRAY A B C BASE)
        (LIST (TAG 'NAT
                   (BOOL-TO-NAT
                    (UNTAG (BIG-ADD-CARRY-OUT A B C BASE)))))).
```

`BOOL-TO-NAT` (page 97) is defined as part of the formal definition of Piton and converts the Piton Booleans `T` and `F` to 1 and 0, respectively. On the two input big numbers shown above, `BIG-PLUS` returns `'((NAT 98) (NAT 6) (NAT 35) (NAT 82) (NAT 0) (NAT 0))`. Note the extra high order 0 indicating that carry out did not occur.

Calling this the ‘‘sum’’ of the two big numbers is justified by the observation that the natural represented by the big number sum of `A` and `B` is the Peano sum of the naturals represented by `A` and `B`. This holds for all big numbers `A` and `B` of equal length, provided the base is a natural number greater than 1. The formal rendering of this general remark is

Theorem. Numeric Interpretation of Big Number Addition.

```
(IMPLIES (AND (BIGNP A BASE)
              (BIGNP B BASE)
              (EQUAL (LENGTH A) (LENGTH B))
              (NUMBERP BASE)
              (LESSP 1 BASE))
         (EQUAL (BIGN->NAT (BIG-PLUS A B C BASE)
                           BASE)
                (PLUS (BIGN->NAT A BASE)
                      (BIGN->NAT B BASE)
                      (TV->NAT C)))).
```

We have proved the theorem above mechanically.

This concludes the formal discussion of the abstract concept of big number addition. We can summarize this section as follows. We defined what a big number (in a given base) is. We defined the natural represented by a given big number. We defined the big number sum of two big numbers. We justified the use of the word ‘‘sum’’ by relating big number addition to Peano addition.

3.3. A Piton Program for Big Number Addition

Our objective is to implement a Piton program that computes **BIG-ADD-ARRAY** and **BIG-ADD-CARRY-OUT** in the special case in which the input carry flag is **F**. We are not interested in implementing **BIG-PLUS** because our envisioned applications of big number arithmetic use fixed length big numbers. For our purposes, **BIG-PLUS** is simply a mathematical abstraction that is useful in justifying our interest in **BIG-ADD-ARRAY** and **BIG-ADD-CARRY-OUT**. In Figure 3-1 we show the Piton program named **BIG-ADD**. It expects three arguments: **A**, the address of the least significant digit in the first big number array; **B**, the address of the least significant digit in the second big number array; and **N**, the length of the two big number arrays. The base of the big number system is implicitly the first unrepresentable natural on the Piton machine. The subroutine sums the two arrays, overwriting the first big number. It leaves a Piton Boolean on the stack indicating whether the sum ‘‘carried out’’ of the array. More precisely, at the conclusion of the program, the data area addressed by the input value of **A** will contain the successive digits of **BIG-ADD-ARRAY** and on top of the temporary stack we will find **BIG-ADD-CARRY-OUT**. We will exhibit a formal specification of this program later.

3.4. An Initial State for Big Number Addition

Most of the work of specifying **BIG-ADD** was done when we defined **BIGNP**, **BIG-ADD-ARRAY** and **BIG-ADD-CARRY-OUT** and such English phrases as ‘‘big number addition.’’ It remains however to cast into a formula the remark that **BIG-ADD** ‘‘adds the two big numbers together, overwrites the first, and leaves the carry out flag on the stack.’’ This necessarily involves the notions of Piton states, the Piton interpreter, resource errors, etc. Before we embark on this formalization we simply illustrate the behavior of **BIG-ADD**—and in so doing familiarize the reader with the structure of Piton states.

To execute **BIG-ADD** we will call it from the **MAIN** program shown below. The program assumes that the data segment of our initial p-state contains at least four data areas: arrays **BNA** and **BNB** (‘‘Big Number **A**’’ and ‘‘Big Number **B**’’), each of which is of length n, a global variable **N**, whose value is n, and another global variable **C**. **MAIN** pushes the starting address of both **BNA** and **BNB** onto the stack, pushes their length on the stack, and calls **BIG-ADD**. The call will overwrite **BNA**. Upon termination of **BIG-ADD**,

Figure 3-1: A Piton Program for Big Number Addition

```
(BIG-ADD    (A B N)                          ; Formal parameters
            NIL                              ; Temporary variables
                                             ; Body
            (PUSH-CONSTANT (BOOL F))         ; Push the input carry flag for
                                             ; the first ADD-NAT-WITH-CARRY
            (PUSH-LOCAL A)                   ; Push the address A

   (DL LOOP ()                               ; This is the top level loop.
                                             ; Every time we get here the carry
                                             ; flag from the last addition and
                                             ; the current value of A will be
                                             ; on the stack.
            (FETCH))                         ; Fetch next digit from A
            (PUSH-LOCAL B)                   ; Push the address B
            (FETCH)                          ; Fetch next digit from B
            (ADD-NAT-WITH-CARRY)             ; Add the two digits and flag
            (PUSH-LOCAL A)                   ; Deposit the sum digit in A
            (DEPOSIT)                        ; (but leave carry flag)
            (PUSH-LOCAL N)                   ; Decrement N by 1
            (SUB1-NAT)
            (SET-LOCAL N)                    ; (but leave N on the stack)
            (TEST-NAT-AND-JUMP ZERO DONE)    ; If N=0, go to DONE
            (PUSH-LOCAL B)                   ; Increment B by 1
            (PUSH-CONSTANT (NAT 1))
            (ADD-ADDR)
            (POP-LOCAL B)
            (PUSH-LOCAL A)                   ; Increment A by 1
            (PUSH-CONSTANT (NAT 1))
            (ADD-ADDR)
            (SET-LOCAL A)                    ; (but leave A on the stack)
            (JUMP LOOP)                      ; goto LOOP
   (DL DONE ()
            (RET)))                          ; Exit.
```

MAIN pops the output carry flag off the temporary stack and into the global variable **C** and halts. **MAIN** has no formals and no temporary variables.

```
(MAIN NIL NIL
      (PUSH-CONSTANT (ADDR (BNA . 0)))
      (PUSH-CONSTANT (ADDR (BNB . 0)))
      (PUSH-GLOBAL N)
      (CALL BIG-ADD)
      (POP-GLOBAL C)
      (RET))
```

Suppose we wished to use **MAIN** to solve the big number version of

$$\begin{array}{r l} & 786{,}433{,}689{,}351{,}873{,}913{,}098{,}236{,}738 \\ + & \underline{141{,}915{,}430{,}937{,}733{,}100{,}148{,}932{,}872} \\ & ? \end{array}$$

In base 2^{32} (which is 4,294,967,296) these two naturals can be represented by the following big numbers of length 4:

'((NAT 246838082) (NAT 3116233281) (NAT 42632655) (NAT 0))

and

'((NAT 3579363592) (NAT 3979696680) (NAT 7693250) (NAT 0)).

A suitable Piton initial state for adding together these two big numbers is shown in Figure 3-2.

The nine fields of the p-state in Figure 3-2 are enumerated and named in the comments of the figure. We discuss each field in turn. Field (1) is the program counter. Note that it is a tagged address. The **PC** tag indicates that it is an address into the program segment. The pair **(MAIN . 0)** is an address, pointing to the the 0^{th} instruction in the **MAIN** program. Field (2) is the control stack. In this example it contains only one frame and so is of the form **'((bindings return-pc))**. Since the current program counter is in **MAIN**, the single frame on the control stack describes the invocation of **MAIN**. Since **MAIN** has no local variables, the frame has the empty list, **NIL**, as the local variable bindings. Since there is only one frame on the stack, it describes the top-level entry into Piton and hence the return program counter is completely irrelevant. If control is ever ‘‘returned’’ from this invocation of **MAIN** the Piton machine will halt rather than ‘‘return control’’ outside of Piton. However, despite the fact that the initial return program counter is irrelevant we insist that it be a legal program counter and so in this example we chose **(PC (MAIN . 0))**. Field (3) is the temporary stack. In this example it is empty. Field (4) is the program segment. It contains two programs, **MAIN** and **BIG-ADD**. Field (5) is the data segment. It contains four ‘‘global arrays’’ named, respectively, **BNA**, **BNB**, **N** and **C**. **BNA** and **BNB** are both arrays of length four. **N** and **C** are each arrays of length one. We think of **N** and **C** simply as global variables. The **BNA** array contains the first of the two big numbers we wish to add, namely **((NAT 246838082) (NAT 3116233281) (NAT 42632655) (NAT 0))**. The **BNB** array contains the second big number. **N** contains the (tagged) length of the two arrays. **C** contains the (tagged) natural number 0; the initial value of **C** is irrelevant however. Fields (6)-(8) are, respectively, the maximum control stack size, 10, the maximum temporary stack size, 8, and the word size, 32. The stack sizes declared in this example are unusually small but sufficient for the computation described. Finally, field (9) is the program status word **RUN**.

Let $\mathbf{p_0}$ be the p-state in Figure 3-2. If one steps this p-state forward 76 times the result is the p-state shown in Figure 3-3. That is, the p-state in Figure 3-3 is equal to **(P** $\mathbf{p_0}$ **76)**.

Observe that the psw in Figure 3-3 is **HALT**. This tells us the computation terminated without error. Because the Piton interpreter is a no-op on states with psw **HALT**, we would get the same result had we stepped $\mathbf{p_0}$ more than 76 times. The program counter points to the **RET** statement in the **MAIN** program, the last instruction executed. The control stack and the temporary stack are exactly as they were in the initial state. The program segment and resource limits are exactly as they were in the initial state—they are never changed. The final value of the **A** array in the data segment is now the big number **((NAT 3826201674) (NAT 2800962665) (NAT 50325906) (NAT 0))**. The final value of the global variable **C** is the Boolean value **F**, indicating that the addition did not carry out of the array. A little arithmetic will confirm that the natural represented by the final values of **BNA** and **C** is 928,349,120,289,607,013,247,169,610, which is the sum of 786,433,689,351,873,913,098,236,738 and 141,915,430,937,733,100,148,932,872, as desired.

3.5. The Formal Specification of BIG-ADD

We now develop a formula that expresses the idea that **BIG-ADD** computes the big number sum of its two arguments, i.e., overwrites its first argument with the **BIG-ADD-ARRAY** and pushes

Figure 3-2: An Initial Piton State for Big Number Addition

```
(P-STATE '(PC (MAIN . 0))                        ; (1) program counter
         '((NIL (PC (MAIN . 0))))                ; (2) control stack
         NIL                                     ; (3) temporary stack

         '((MAIN NIL NIL                         ; (4) program segment
                 (PUSH-CONSTANT (ADDR (BNA . 0)))
                 (PUSH-CONSTANT (ADDR (BNB . 0)))
                 (PUSH-GLOBAL N)
                 (CALL BIG-ADD)
                 (POP-GLOBAL C)
                 (RET))
           (BIG-ADD (A B N) NIL
                 (PUSH-CONSTANT (BOOL F))
                 (PUSH-LOCAL A)
             (DL LOOP NIL (FETCH))
                 (PUSH-LOCAL B)
                 (FETCH)
                 (ADD-NAT-WITH-CARRY)
                 (PUSH-LOCAL A)
                 (DEPOSIT)
                 (PUSH-LOCAL N)
                 (SUB1-NAT)
                 (SET-LOCAL N)
                 (TEST-NAT-AND-JUMP ZERO DONE)
                 (PUSH-LOCAL B)
                 (PUSH-CONSTANT (NAT 1))
                 (ADD-ADDR)
                 (POP-LOCAL B)
                 (PUSH-LOCAL A)
                 (PUSH-CONSTANT (NAT 1))
                 (ADD-ADDR)
                 (SET-LOCAL A)
                 (JUMP LOOP)
             (DL DONE NIL (RET))))

         '((BNA  (NAT 246838082)                 ; (5) data segment
                 (NAT 3116233281)
                 (NAT 42632655)
                 (NAT 0))
           (BNB  (NAT 3579363592)
                 (NAT 3979696680)
                 (NAT 7693250)
                 (NAT 0))
           (N    (NAT 4))
           (C    (NAT 0)))

         10                                      ; (6) max ctrl stk size
         8                                       ; (7) max temp stk size
         32                                      ; (8) word size
         'RUN)                                   ; (9) psw
```

Figure 3-3: A Final Piton State for Big Number Addition

```
(P-STATE '(PC (MAIN . 5))                          ; program counter
         '((NIL (PC (MAIN . 0))))                  ; control stack
         NIL                                       ; temporary stack
         '((MAIN NIL NIL                           ; program segment
                 (PUSH-CONSTANT (ADDR (BNA . 0)))
                 (PUSH-CONSTANT (ADDR (BNB . 0)))
                 (PUSH-GLOBAL N)
                 (CALL BIG-ADD)
                 (POP-GLOBAL C)
                 (RET))
           (BIG-ADD (A B N) NIL
                 (PUSH-CONSTANT (BOOL F))
                 (PUSH-LOCAL A)
             (DL LOOP NIL (FETCH))
                 (PUSH-LOCAL B)
                 (FETCH)
                 (ADD-NAT-WITH-CARRY)
                 (PUSH-LOCAL A)
                 (DEPOSIT)
                 (PUSH-LOCAL N)
                 (SUB1-NAT)
                 (SET-LOCAL N)
                 (TEST-NAT-AND-JUMP ZERO DONE)
                 (PUSH-LOCAL B)
                 (PUSH-CONSTANT (NAT 1))
                 (ADD-ADDR)
                 (POP-LOCAL B)
                 (PUSH-LOCAL A)
                 (PUSH-CONSTANT (NAT 1))
                 (ADD-ADDR)
                 (SET-LOCAL A)
                 (JUMP LOOP)
             (DL DONE NIL (RET))))
         '((BNA  (NAT 3826201674)                  ; data segment
                 (NAT 2800962665)
                 (NAT 50325906)
                 (NAT 0))
           (BNB  (NAT 3579363592)
                 (NAT 3979696680)
                 (NAT 7693250)
                 (NAT 0))
           (N    (NAT 4))
           (C    (BOOL F)))
         10                                        ; max ctrl stk size
          8                                        ; max temp stk size
         32                                        ; word size
         'HALT)                                    ; psw
```

BIG-ADD-CARRY-OUT onto the stack.

3.5.1. Preliminary Definitions

We will need to talk about the arrays associated with given data area names in a given data segment. We will also need to discuss the data segment obtained from another by changing the array associated with a given name. These concepts are easily expressed in terms of functions defined in the formalization of Piton (Chapter 7) but because the names used there are unfamiliar we will define slightly more memorable names here.

Definition.
(ARRAY NAME SEGMENT) = **(CDR (ASSOC NAME SEGMENT))**

defines the function that returns the array associated with **NAME** in a given data segment **SEGMENT**. **ASSOC** is a primitive function in our logic.

Definition.
(PUT-ARRAY A NAME SEGMENT) = **(PUT-ASSOC A NAME SEGMENT)**

defines the function that returns a new data segment obtained from **SEGMENT** by associating the array **A** with data area name **NAME** and leaving all other data areas unchanged. **PUT-ASSOC** is defined on page 151.

So that we can describe the program segment succinctly we will define the constant function **BIG-ADD-PROGRAM** to be the list constant corresponding to Figure 3-1, page 29.

Definition.

```
(BIG-ADD-PROGRAM)
   =
'(BIG-ADD    (A B N)                         ; Formal parameters
             NIL                             ; Temporary variables
                                             ; Body
             (PUSH-CONSTANT (BOOL F))        ; Push the input carry flag for
             ...                             ; ...

    (DL DONE ()
             (RET))).                        ; Exit.
```

3.5.2. The Initial State

To develop the specification of **BIG-ADD** we will consider an ‘‘arbitrary’’ initial p-state in which the current instruction is a legal **CALL** of **BIG-ADD** and we will describe the final p-state produced by executing that **CALL** statement and all of the **BIG-ADD** computation up to and including the return.

The ‘‘arbitrary’’ initial state will be

```
(P-STATE PC
         CTRL-STK
         (APPEND (LIST (TAG 'NAT N)
                       (TAG 'ADDR (CONS B 0))
                       (TAG 'ADDR (CONS A 0)))
                 TEMP-STK)
         PROG-SEGMENT
         DATA-SEGMENT
         MAX-CTRL-STK-SIZE
         MAX-TEMP-STK-SIZE
         WORD-SIZE
         'RUN),
```

where **PC** is assumed to point to the instruction **(CALL BIG-ADD)** and **BIG-ADD** is defined as in Figure 3-1. Let **P0** be the p-state above. Then the additional constraints mentioned can be formalized by saying

```
(EQUAL (P-CURRENT-INSTRUCTION P0) '(CALL BIG-ADD))
```

and

```
(EQUAL (DEFINITION 'BIG-ADD PROG-SEGMENT)
       (BIG-ADD-PROGRAM)).
```

Observe that the psw of **P0** is **'RUN**. Note also that that the temporary stack in **P0** consists of some arbitrary **TEMP-STK** with three additional items pushed onto it. The items (in the order in which they were pushed) are a tagged data address to location 0 of the data area **A**, a tagged data address to location 0 of the data area **B**, and a tagged natural **N**. Note carefully that the ''**A**,'' ''**B**,'' and ''**N**''used here are variable symbols which are so far unconstrained. In summary, **P0** is an arbitrary p-state poised to execute a **CALL** of our **BIG-ADD** on three arguments that are tagged in accordance with our expectations. However, much more needs to said about those arguments.

3.5.3. The Preconditions

The ''expected'' values of **A** and **B** are the names of data areas that contain big numbers of equal length and the ''expected'' value of **N** is the length of those big numbers. These ''expectations'' are part of the ''input conditions'' of **BIG-ADD** (which heretofore have been implicit in our discussions) and can be expressed formally as the conjunction of

```
(DEFINEDP A (P-DATA-SEGMENT P0))
(DEFINEDP B (P-DATA-SEGMENT P0))
(NOT (EQUAL A B))
(BIGNP (ARRAY A (P-DATA-SEGMENT P0)) (EXP 2 (P-WORD-SIZE P0)))
(BIGNP (ARRAY B (P-DATA-SEGMENT P0)) (EXP 2 (P-WORD-SIZE P0)))
(EQUAL N (LENGTH (ARRAY A (P-DATA-SEGMENT P0))))
```

and

```
(EQUAL N (LENGTH (ARRAY B (P-DATA-SEGMENT P0)))).
```

The function **P-DATA-SEGMENT** is one of the accessors of the **P-STATE** shell constructor and returns the data segment of the state (see page 131). The function **DEFINEDP** checks that its first argument is the name of a data area in its second argument and is defined on page 97. Observe that we explicitly assume that **A** is different from **B**, i.e., that the program is operating on two distinct data areas. (Of course, they may contain the same big number.) This is not technically necessary; **BIG-ADD** performs in a meaningful way even if it is passed the same address in both arguments. But by ruling out this possibility we simplify our analysis of the program somewhat.

The above conditions are the obvious preconditions for **BIG-ADD**. However, if **BIG-ADD** is to run without error there are several more details we must consider. We address them in roughly the order in which they arise in the execution of the code for **BIG-ADD**.

In order for the **CALL** statement to execute without error, we must know that there is enough room on the control stack to build the new frame. Inspection of the definition of **P-CTRL-STK-SIZE** (page 108) shows that the frame we will build has size five (three for the local variables of **BIG-ADD** plus two more). Thus, we must assume

```
(NOT (LESSP (P-MAX-CTRL-STK-SIZE P0)
            (PLUS 5 (P-CTRL-STK-SIZE (P-CTRL-STK P0))))).
```

We must similarly worry about the temporary stack overflowing during the **BIG-ADD** computation. Once we enter **BIG-ADD** the temporary stack will be **TEMP-STK** (because the three actuals will have been popped off). By how far will we extend **TEMP-STK** during the computation? Inspection will show that we need at most three more slots. For example, once we have executed the first **(PUSH-LOCAL B)** we have pushed three items onto the stack. At no point do we have more than three items pushed. It is just a coincidence that the amount of temporary stack needed by **BIG-ADD**'s body is the same as the number of actuals on the stack at the time of the **CALL**. In any case, we must know

```
(NOT (LESSP (P-MAX-TEMP-STK-SIZE P0)
            (PLUS 3 (LENGTH (P-TEMP-STK P0))))).
```

If this assumption is violated, some push in the body of **BIG-ADD** causes a stack overflow error.

Once we enter the **LOOP** in **BIG-ADD** we fetch the first digit from **A**. But this assumes there is a first digit, i.e., that the argument arrays have nonzero length. **BIG-ADD** could have been coded to work for empty big numbers, by checking **N** before the first **FETCH**. But as it is coded, **BIG-ADD** assumes the big numbers are nonempty and does not check **N** until it has added the low order digits together and decremented **N**. Thus, we must assume

```
(NOT (ZEROP N)).
```

If this assumption is violated, **BIG-ADD** causes an addressing error.

When we decrement **N** with **SUB1-NAT** we must know that **N** is a representable natural. Thus,

```
(LESSP N (EXP 2 (P-WORD-SIZE P0))).
```

If this assumption is violated, **BIG-ADD** causes an arithmetic error. We can prove from the foregoing assumptions that we will not get addressing errors when we increment **A** and **B** **N** times.

Finally, when we execute the **RET** statement at the conclusion of **BIG-ADD** we must know that the initial control stack was nonempty. Thus,

```
(LISTP (P-CTRL-STK P0)).
```

No Piton state should have an empty control stack.

For convenience, we collect all of these conditions together into a single predicate.

Definition .

```
(BIG-ADD-INPUT-CONDITIONP A B N P0)
   =
(AND (DEFINEDP A (P-DATA-SEGMENT P0))
     (DEFINEDP B (P-DATA-SEGMENT P0))
     (NOT (EQUAL A B))
     (BIGNP (ARRAY A (P-DATA-SEGMENT P0)) (EXP 2 (P-WORD-SIZE P0)))
     (BIGNP (ARRAY B (P-DATA-SEGMENT P0)) (EXP 2 (P-WORD-SIZE P0)))
     (EQUAL N (LENGTH (ARRAY A (P-DATA-SEGMENT P0))))
     (EQUAL N (LENGTH (ARRAY B (P-DATA-SEGMENT P0))))
     (NOT (LESSP (P-MAX-CTRL-STK-SIZE P0)
                 (PLUS 5 (P-CTRL-STK-SIZE (P-CTRL-STK P0)))))
     (NOT (LESSP (P-MAX-TEMP-STK-SIZE P0)
                 (PLUS 3 (LENGTH (P-TEMP-STK P0)))))
     (NOT (ZEROP N))
     (LESSP N (EXP 2 (P-WORD-SIZE P0)))
     (LISTP (P-CTRL-STK P0)))
```

In our correctness theorem we will assume that **(BIG-ADD-INPUT-CONDITIONP A B N P0)** holds.

3.5.4. The Final State

Next we wish to characterize the state obtained by executing the above **CALL** of **BIG-ADD**. Because of the constructive nature of our logic, it will be necessary to say how many instructions we wish to execute.[8] For the moment though, let **clock** stand for some expression that determines the number of Piton instructions executed from the **CALL** above to the **RET** instruction at the end of that invocation of **BIG-ADD**, inclusive. We will define **clock** in the next section. Then the final state obtained by executing **clock** instructions starting at **P0** is **(P P0 clock)**. We would like to characterize **(P P0 clock)** completely.

The program counter of the final state will be one greater than **PC**. The control stack will be exactly the control stack of **P0**. The temporary stack will be **TEMP-STK** with one additional item pushed onto it. The item will be

```
(BIG-ADD-CARRY-OUT (ARRAY A DATA-SEGMENT)
                   (ARRAY B DATA-SEGMENT)
                   F
                   (EXP 2 WORD-SIZE)).
```

The program segment of the final state will be the same as the program segment of **P0**. The data segment of the final state will be the same as the data segment of **P0** with one exception: the array associated with the data area **A** will be

```
(BIG-ADD-ARRAY (ARRAY A DATA-SEGMENT)
               (ARRAY B DATA-SEGMENT)
               F
               (EXP 2 WORD-SIZE)).
```

The resource limitations of the final state will be the same as those of the initial state and the psw will still be **'RUN**. Thus, **(P P0 clock)** is equal to

[8] This is not strictly true. It is possible to phrase partial correctness theorems by including among the hypotheses the assumption that **K** is a number such that the psw of **(P P0 K)** is **HALT**. We do not pursue this here.

```
(P-STATE (ADD1-ADDR PC)
         CTRL-STK
         (PUSH (BIG-ADD-CARRY-OUT (ARRAY A DATA-SEGMENT)
                                  (ARRAY B DATA-SEGMENT)
                                  F
                                  (EXP 2 WORD-SIZE))
               TEMP-STK)
         PROG-SEGMENT
         (PUT-ARRAY (BIG-ADD-ARRAY (ARRAY A DATA-SEGMENT)
                                   (ARRAY B DATA-SEGMENT)
                                   F
                                   (EXP 2 WORD-SIZE))
                    A
                    DATA-SEGMENT)
         MAX-CTRL-STK-SIZE
         MAX-TEMP-STK-SIZE
         WORD-SIZE
         'RUN).
```

The function **ADD1-ADDR** is defined on page 95 as part of the formalization of Piton.

3.5.5. The Clock

It remains only to say what **clock** is. We derive a function, **BIG-ADD-CLOCK**, which determines the number of instructions necessary to execute any legal **CALL** of **BIG-ADD**. In the case of **BIG-ADD** this derived function is a function only of the length, **N**, of the big numbers being added. The **CALL** itself costs one instruction. Inspection of the the code for **BIG-ADD** (page 29) shows that we then execute two more instructions before arriving at the label **LOOP**. Suppose that **BIG-ADD-LOOP-CLOCK** is defined to be the number of instructions it takes to complete the loop and return. Then

Definition.
```
(BIG-ADD-CLOCK N) = (PLUS 3 (BIG-ADD-LOOP-CLOCK N)).
```

To define **BIG-ADD-LOOP-CLOCK** we walk through the code symbolically again. If **N** is 1 we execute 11 instructions from **LOOP** through the **RET** at **DONE**. If **N** is not 1, we execute 19 instructions and arrive back at **LOOP** with **N** decremented by 1. Thus, a suitable definition of **BIG-ADD-LOOP-CLOCK** is:

Definition.
```
(BIG-ADD-LOOP-CLOCK N)
   =
(IF (ZEROP N)
    0
    (IF (EQUAL N 1)
        11
        (PLUS 19 (BIG-ADD-LOOP-CLOCK (SUB1 N))))).
```

The case in which **N** is 0 never arises but is included in the above definition to insure that the function defined is total. Of course, it is easy to see that **(BIG-ADD-LOOP-CLOCK N)** is 11+19(**N-1**), when **N** is non-0. While such an algebraic expression of the clock is pleasing, we prefer the recursive formulation in general because it mirrors the exploration of the code and more easily accomodates special cases (e.g., interior branches).

3.5.6. The Correctness Theorem

The correctness theorem for **BIG-ADD** can now be written down completely. It is shown in Figure 3-4 and should be self-explanatory. The formula is a theorem. It can be proved from the foregoing definitions and the formal definition of Piton. In fact, we have proved it mechanically. We discuss the proof later.

Figure 3-4: The Specification of BIG-ADD

Theorem. Correctness of **BIG-ADD.**

```
(IMPLIES (AND (EQUAL P0 (P-STATE PC
                                 CTRL-STK
                                 (APPEND (LIST (TAG 'NAT N)
                                               (TAG 'ADDR (CONS B 0))
                                               (TAG 'ADDR (CONS A 0)))
                                         TEMP-STK)
                                 PROG-SEGMENT
                                 DATA-SEGMENT
                                 MAX-CTRL-STK-SIZE
                                 MAX-TEMP-STK-SIZE
                                 WORD-SIZE
                                 'RUN))
              (EQUAL (P-CURRENT-INSTRUCTION P0) '(CALL BIG-ADD))
              (EQUAL (DEFINITION 'BIG-ADD PROG-SEGMENT)
                     (BIG-ADD-PROGRAM))
              (BIG-ADD-INPUT-CONDITIONP A B N P0))
         (EQUAL (P P0 (BIG-ADD-CLOCK N))
                (P-STATE (ADD1-ADDR PC)
                         CTRL-STK
                         (PUSH (BIG-ADD-CARRY-OUT (ARRAY A DATA-SEGMENT)
                                                  (ARRAY B DATA-SEGMENT)
                                                  F
                                                  (EXP 2 WORD-SIZE))
                               TEMP-STK)
                         PROG-SEGMENT
                         (PUT-ARRAY
                              (BIG-ADD-ARRAY (ARRAY A DATA-SEGMENT)
                                             (ARRAY B DATA-SEGMENT)
                                             F
                                             (EXP 2 WORD-SIZE))
                              A
                              DATA-SEGMENT)
                         MAX-CTRL-STK-SIZE
                         MAX-TEMP-STK-SIZE
                         WORD-SIZE
                         'RUN)))
```

The theorem of Figure 3-4 is very powerful. It can be applied to any legal call of **BIG-ADD**, no matter what other programs are in the program segment and no matter what data areas are defined in the data segment. It specifies exactly how many instructions will be executed on behalf of the call. For example, to add together two big numbers of length 4 will take 71 Piton instructions. To add together two big numbers of length 100 will take 1,895 Piton instructions. The theorem tells us exactly how to obtain the final state: pop the arguments off the stack, push the output carry on the stack, deposit the big number sum in the data area of the first argument, and keep the psw **RUN**. Note that since we know the final psw is **RUN** we know that no run-time errors occur if the preconditions are satisfied. The beauty of the theorem in Figure 3-4 is that all intents and purposes it allows us to treat **(CALL BIG-ADD)** as a Piton primitive. We illustrate this in the next section.

3.6. Using The Correctness Theorem

In this section we discuss how to ‘‘stack’’ correctness proofs for Piton programs. We explain how the above theorem about **BIG-ADD** can be used to construct a correctness proof for a program that uses **BIG-ADD**. This may help some readers understand why we chose the above form for our correctness theorem. In addition, it suggests that we can verify systems of Piton programs by verifying the individual subroutines—which of course we can but which we have not yet done for systems of nontrivial complexity. We here define a trivial two-layered ‘‘system,’’ consisting simply of **BIG-ADD** and a top-level main program which uses **BIG-ADD**, together with a particular data segment. We prove the correctness of that system, assuming the above theorem about **BIG-ADD**. The style of our proof illustrates how to go about proving the correctness of any program (top-level or not) which uses **BIG-ADD**.

Recall the **MAIN** program on page 29 which uses **BIG-ADD** to add **BNA** and **BNB**. Define the constant function **MAIN-PROGRAM** to be equal to the list constant describing **MAIN**.

Definition.
```
(MAIN-PROGRAM)
   =
'(MAIN       NIL
             NIL
             (PUSH-CONSTANT (ADDR (BNA . 0)))
             (PUSH-CONSTANT (ADDR (BNB . 0)))
             (PUSH-GLOBAL N)
             (CALL BIG-ADD)
             (POP-GLOBAL C)
             (RET)).
```

Our earlier use of **MAIN** was in Figure 3-2 (page 31) where it was the top-level program in a p-state configured to add together two specific big numbers.

Now consider the function

Definition.
```
(SYSTEM-INITIAL-STATE A B)
   =
(P-STATE '(PC (MAIN . 0))
         '((NIL (PC (MAIN . 0))))
         NIL
         (LIST (MAIN-PROGRAM)
               (BIG-ADD-PROGRAM))
         (LIST (CONS 'BNA A)
               (CONS 'BNB B)
               (CONS 'N (LIST (TAG 'NAT (LENGTH A))))
               (CONS 'C (LIST (TAG 'NAT 0))))
         10
         8
         32
         'RUN).
```

If given two big numbers (base 2^{32}) of equal length, this function creates an initial p-state suitable for adding them together and storing the answers in **BNA** and **C**. The p-state shown in Figure 3-2 was actually created by applying **SYSTEM-INITIAL-STATE** to the particular big numbers used in the earlier example. We would like to prove the correctness of the system of Piton programs and data described by **SYSTEM-INITIAL-STATE**.

We claim that the state produced by **SYSTEM-INITIAL-STATE** is suitable for adding big numbers together provided **A** and **B** are both nonempty big numbers (base 2^{32}) of length n, where n is less than 2^{32}. We define the following predicate to check these conditions.

Definition.
```
(SYSTEM-INITIAL-STATE-OKP A B)
    =
(AND (BIGNP A (EXP 2 32))
     (BIGNP B (EXP 2 32))
     (EQUAL (LENGTH A) (LENGTH B))
     (NOT (ZEROP (LENGTH A)))
     (LESSP (LENGTH A) (EXP 2 32)))
```

How long does it take for this system to run to completion? The answer is provided by the function

Definition.
```
(SYSTEM-INITIAL-STATE-CLOCK A B)
    =
(PLUS 5 (BIG-ADD-CLOCK (LENGTH A))).
```

That is, **MAIN** executes five instructions in addition to the **CALL** of **BIG-ADD**.

The following formula describes the correctness of the system:

Theorem. Correctness of a **BIG-ADD** System
```
(IMPLIES
 (SYSTEM-INITIAL-STATE-OKP A B)
 (EQUAL (P (SYSTEM-INITIAL-STATE A B)
           (SYSTEM-INITIAL-STATE-CLOCK A B))
        (P-STATE
         '(PC (MAIN . 5))
         '((NIL (PC (MAIN . 0))))
         NIL
         (LIST (MAIN-PROGRAM)
               (BIG-ADD-PROGRAM))
         (LIST (CONS 'BNA (BIG-ADD-ARRAY A B F (EXP 2 32)))
               (CONS 'BNB B)
               (CONS 'N (LIST (TAG 'NAT (LENGTH A))))
               (CONS 'C (LIST (BIG-ADD-CARRY-OUT A B F (EXP 2 32)))))
         10 8 32 'HALT))).
```

That is, if **A** and **B** satisfy **SYSTEM-INITIAL-STATE-OKP** then the result of running **SYSTEM-INITIAL-STATE** for **SYSTEM-INITIAL-STATE-CLOCK** instructions is a **HALT**ed p-state in which the array named **BNA** has the value **(BIG-ADD-ARRAY A B F (EXP 2 32))** and the global variable **C** has the value **(BIG-ADD-CARRY-OUT A B F (EXP 2 32))**.

To prove this we must first observe an important theorem about the Piton interpreter **P**.

Theorem. Sequential Execution.
```
(EQUAL (P S (PLUS I J))
       (P (P S I) J))
```

This theorem says that running forward **I**+**J** steps is the same as running forward **I** steps and then running forward **J** steps from there. The proof is trivial by induction on **S** and **I**.

We now present the proof of the correctness of our **BIG-ADD** system. Readers interested in constructing the proof formally should first read Chapter 7, where we present the formal definition of Piton. Because this is the first Piton proof we have discussed, we will go very slowly. The proof is in fact immediate from the correctness of **BIG-ADD**.

Proof. We will prove the correctness of our **BIG-ADD** system by reducing the left-hand side of the conclusion, which we'll call $\mathbf{p_{left}}$, to the right-hand side, which we'll call $\mathbf{p_{right}}$. Let $\mathbf{p_0}$ be **(SYSTEM-INITIAL-STATE A B)**. Then $\mathbf{p_{left}}$ is

(P $\mathbf{p_0}$ **(SYSTEM-INITIAL-STATE-CLOCK A B)).**

But **(SYSTEM-INITIAL-STATE-CLOCK A B)** is 5+**(BIG-ADD-CLOCK (LENGTH A))**, which can be written as 3+**(BIG-ADD-CLOCK (LENGTH A))**+2. Thus, by the sequential execution theorem, $\mathbf{p_{left}}$ can be equivalently obtained by composing three smaller runs of **P**. The first run, which we will call *Run 1* starts from $\mathbf{p_0}$, takes three steps, and produces a state we'll call $\mathbf{p_3}$. The second run, *Run 2*, starts at $\mathbf{p_3}$, takes **(BIG-ADD-CLOCK (LENGTH A))** steps, and produces a state we'll call $\mathbf{p_{3+c}}$. The third run, *Run 3*, starts at $\mathbf{p_{3+c}}$, takes two steps, and produces a state we'll call $\mathbf{p_{done}}$. $\mathbf{p_{left}}$ is equal to $\mathbf{p_{done}}$ by the sequential execution lemma. It will turn out that $\mathbf{p_{done}}$ is identical to $\mathbf{p_{right}}$, as desired. We proceed by working on each of the three runs in turn.

Run 1. Our starting state, $\mathbf{p_0}$, is **(SYSTEM-INITIAL-STATE A B)**, which is

```
(P-STATE '(PC (MAIN . 0))
         '((NIL (PC (MAIN . 0))))
         NIL
         (LIST (MAIN-PROGRAM)
               (BIG-ADD-PROGRAM))
         (LIST (CONS 'BNA A)
               (CONS 'BNB B)
               (CONS 'N (LIST (TAG 'NAT (LENGTH A))))
               (CONS 'C (LIST (TAG 'NAT 0))))
         10
         8
         32
         'RUN)
```

by the definition of **SYSTEM-INITIAL-STATE**. We wish to obtain $\mathbf{p_3}$, which is **(P** $\mathbf{p_0}$ **3)**. Recall how **P** is defined: if the precondition of the current instruction is satisfied, the next state is that obtained by applying the current step function to the old state. The program counter of $\mathbf{p_0}$ points to the 0^{th} instruction of **MAIN**. Thus, the current instruction is **(PUSH-CONSTANT (ADDR (BNA . 0)))**. The preconditions of **PUSH-CONSTANT** are satisfied: there is room to push at least one item on the temporary stack because the maximum stack length is 8 and the stack is currently empty. Thus, **(P** $\mathbf{p_0}$ **3)** is equal to **(P** $\mathbf{p_1}$ **2)**, where $\mathbf{p_1}$ is the result of stepping $\mathbf{p_0}$ with the **PUSH-CONSTANT** step function. The **PUSH-CONSTANT** step function increments the program counter by one and pushes the given constant onto the temporary stack. Thus, $\mathbf{p_1}$ is

```
(P-STATE '(PC (MAIN . 1))
         '((NIL (PC (MAIN . 0))))
         '((ADDR (BNA . 0)))
         (LIST (MAIN-PROGRAM)
               (BIG-ADD-PROGRAM))
         (LIST (CONS 'BNA A)
               (CONS 'BNB B)
               (CONS 'N (LIST (TAG 'NAT (LENGTH A))))
               (CONS 'C (LIST (TAG 'NAT 0))))
         10
         8
         32
         'RUN).
```

The derivation of p_1 from p_0 illustrates the most fundamental Piton proof technique: determine the current instruction, check that the precondition is satisfied, and apply the step function. This technique is called *symbolic execution* of Piton.

We can compute **(P p_1 2)** by two more applications of symbolic execution, first running the **PUSH-CONSTANT** instruction at **(PC (MAIN . 1))** (which increments the program counter and pushes the **BNB** address) and then running the **PUSH-GLOBAL** instruction at **(PC (MAIN . 2))** (which increments the program counter and pushes the current value of **N**). The result is

```
(P-STATE '(PC (MAIN . 3))
         '((NIL (PC (MAIN . 0))))
         (LIST (TAG 'NAT (LENGTH A))
               '(ADDR (BNB . 0))
               '(ADDR (BNA . 0)))
         (LIST (MAIN-PROGRAM)
               (BIG-ADD-PROGRAM))
         (LIST (CONS 'BNA A)
               (CONS 'BNB B)
               (CONS 'N (LIST (TAG 'NAT (LENGTH A))))
               (CONS 'C (LIST (TAG 'NAT 0))))
         10 8 32 'RUN).
```

The above p-state is p_3. This completes the first run.

Run 2. We now wish to obtain p_{3+c}, which by definition is **(P p_3 (BIG-ADD-CLOCK (LENGTH A)))**. Observe that the current instruction of p_3 is **(CALL BIG-ADD)**. If we used symbolic execution here we would next investigate the precondition for **CALL** and apply the **CALL** step function, taking us into the body of **BIG-ADD**. That would be a strategic mistake. Instead, we will appeal to the correctness theorem for **BIG-ADD**.

First, observe that the temporary stack of p_3 can be equivalently written as an **APPEND** of three **TAG**ged objects to the empty stack. That is, p_3 above is equivalent to:

```
(P-STATE '(PC (MAIN . 3))
         '((NIL (PC (MAIN . 0))))
         (APPEND (LIST (TAG 'NAT (LENGTH A))
                       (TAG 'ADDR '(BNB . 0))
                       (TAG 'ADDR '(BNA . 0)))
                 NIL)
         (LIST (MAIN-PROGRAM)
               (BIG-ADD-PROGRAM))
         (LIST (CONS 'BNA A)
               (CONS 'BNB B)
               (CONS 'N (LIST (TAG 'NAT (LENGTH A))))
               (CONS 'C (LIST (TAG 'NAT 0))))
         10 8 32 'RUN).
```

But now we see we are in exactly the situation handled by the theorem stating the correctness of **BIG-ADD**: we have a state whose current instruction is a call to our **BIG-ADD** program on legal input and we wish to run forward **(BIG-ADD-CLOCK (LENGTH A))** steps. Thus, $\mathbf{p}_{\mathbf{3+c}}$ is obtained by instantiating the right-hand side of the conclusion of the correctness of **BIG-ADD**,

```
(P-STATE '(PC (MAIN . 4))
         '((NIL (PC (MAIN . 0))))
         (PUSH (BIG-ADD-CARRY-OUT A B F (EXP 2 32))
               NIL)
         (LIST (MAIN-PROGRAM)
               (BIG-ADD-PROGRAM))
         (LIST (CONS 'BNA (BIG-ADD-ARRAY A B F (EXP 2 32)))
               (CONS 'BNB B)
               (CONS 'N (LIST (TAG 'NAT (LENGTH A))))
               (CONS 'C (LIST (TAG 'NAT 0))))
         10 8 32 'RUN)
```

This concludes the second run.

Run 3. We obtain $\mathbf{p}_{\mathbf{done}}$ by stepping forward from $\mathbf{p}_{\mathbf{3+c}}$ two more steps. We use symbolic execution again, running the **POP-GLOBAL** at **(PC (MAIN . 4))** and then the **RET** at **(PC (MAIN . 5))**. The result is

```
(P-STATE '(PC (MAIN . 5))
         '((NIL (PC (MAIN . 0))))
         NIL
         (LIST (MAIN-PROGRAM)
               (BIG-ADD-PROGRAM))
         (LIST (CONS 'BNA (BIG-ADD-ARRAY A B F (EXP 2 32)))
               (CONS 'BNB B)
               (CONS 'N (LIST (TAG 'NAT (LENGTH A))))
               (CONS 'C (BIG-ADD-CARRY-OUT A B F (EXP 2 32))))
         10 8 32 'HALT),
```

which is the desired final state, $\mathbf{p}_{\mathbf{right}}$. **Q.E.D.**

The point of this exercise is that the correctness theorem for **BIG-ADD** permitted us to prove **MAIN** correct without ever considering the code for **BIG-ADD**. Observe that all of the program counters in the states mentioned in the proof above are in **MAIN**. Observe also that our correctness result for **BIG-ADD** readily applied to a state containing another program (i.e., **MAIN**) and data areas irrelevant to **BIG-ADD** (i.e., **C**). Finally, observe that the sequential execution theorem freed us from having to consider exactly how many clock ticks were available when we encountered the call of **BIG-ADD**. If there are at least

(BIG-ADD-CLOCK n), where **n** is the length of the big numbers in question, then we can step past the **CALL** and decrement the clock by **(BIG-ADD-CLOCK n)**. In essence, the correctness theorem for **BIG-ADD** permits us to treat **(CALL BIG-ADD)** as though it were a primitive instruction; in one proof step we obtain the state produced by any successful execution.

3.7. The Proof of the Correctness of BIG-ADD

We sketch the proof of the correctness of the **BIG-ADD** program very briefly. As seen in the proof above, it is relatively straightforward to deal with ‘‘straight line’’ Piton code. One merely symbolically executes the definition of Piton, accumulating into the state the successive changes. **BIG-ADD** is more subtle because it has a loop in it. The presence of the loop immediately suggests induction, which in turn suggests that we need some more general concepts with which to discuss the ‘‘invariants’’ maintained at intermediate arrivals at the top of the loop. So much for generalities.

The key to our proof of the correctness of **BIG-ADD** is to define two functions derived from the loop in **BIG-ADD** that compute the final big number array and the final carry out ‘‘in the same way’’ that the Piton loop does. We call these derived functions **BIG-ADD-ARRAY-LOOP** and **BIG-ADD-CARRY-OUT-LOOP**. We then prove that the derived functions are (a) computed by the loop in **BIG-ADD** and (b) are equivalent to those used in the abstract specification.

3.7.1. The Derived Specification Functions

The following function can be thought of as the essence of the loop in **BIG-ADD** *vis-a-vis* its effect on the array stored in the **A** data area. In the function below, **I** is a natural number that is the location in the **A** array at which the **A** address is pointing. **A-ARRAY** and **B-ARRAY** are the arrays associated with the **A** and **B** data areas of **BIG-ADD**. **N** plays the same role as in **BIG-ADD**, namely, it is the distance from the current location, **I**, to the end of the arrays. That is, **N** is the number of iterations we are to perform. **C** is the truth value corresponding to the current input carry flag and **BASE** is the base of the big number system.

Definition.
```
(BIG-ADD-ARRAY-LOOP I A-ARRAY B-ARRAY N C BASE)
   =
(IF (ZEROP (SUB1 N))
    (PUT (TAG 'NAT
              (REMAINDER (PLUS (UNTAG (GET I A-ARRAY))
                               (UNTAG (GET I B-ARRAY))
                               (TV->NAT C))
                         BASE))
         I
         A-ARRAY)
    (BIG-ADD-ARRAY-LOOP (ADD1 I)
                        (PUT (TAG 'NAT
                                  (REMAINDER
                                   (PLUS (UNTAG (GET I A-ARRAY))
                                         (UNTAG (GET I B-ARRAY))
                                         (TV->NAT C))
                                   BASE))
                             I
                             A-ARRAY)
                        B-ARRAY
                        (SUB1 N)
                        (NOT (LESSP (PLUS (UNTAG (GET I A-ARRAY))
                                          (UNTAG (GET I B-ARRAY))
                                          (TV->NAT C))
                                    BASE))
                        BASE))
```

The function **PUT** is defined on page 151 as part of the definition of Piton. **(PUT val i a)** puts **val** at the **i**th location of the array **a** and returns the resulting array. Observe that **BIG-ADD-ARRAY-LOOP** iterates **N** times, considering successive array positions starting at position **I**, and **PUT**s into each position the corresponding digit of the big number sum of the two arrays.

We also define the analogous derived function which describes the final carry out computed by the loop and call it **BIG-ADD-CARRY-OUT-LOOP**. We do not exhibit its definition here.

These derived functions let us separate the problem of what the Piton program computes from the problem of whether it computes the specified quantities. We discuss these two problems in the next two sections.

3.7.2. The Equivalence of the Derived Functions and the Piton Code

The next step in our proof is to establish that the derived functions indeed describe the effects of the loop in **BIG-ADD**. This immediately raises another Piton specification challenge.

In Figure 3-5 we specify the loop in **BIG-ADD**. The formula, which is very reminiscent of the specification of **BIG-ADD** itself, is of the form **(IMPLIES hyp (EQUAL (P** p_0 **k)** p_k**))** where p_0 is the initial state of a Piton computation of length **k** and p_k is the final state. In this theorem, the initial state describes an ''arbitrary'' legal arrival at **LOOP** in **BIG-ADD** and **k** measures a run up through the execution of the **RET** statement in **BIG-ADD**. Note that the hypothesis is basically that part of **BIG-ADD-INPUT-CONDITIONP** that is needed to get us through the loop without error. The program counter in the initial state is **(PC (BIG-ADD . 2))**, which points to the instruction labelled **LOOP**. The ''shape'' of the control stack in the initial state is as constructed by any legal call of **BIG-ADD**, that is, the

Figure 3-5: The Specification of the Loop in BIG-ADD

Theorem.

```
(IMPLIES
 (AND (LESSP (LENGTH (ARRAY A DATA-SEGMENT)) (EXP 2 WORD-SIZE))
      (NOT (ZEROP WORD-SIZE))
      (LISTP CTRL-STK)
      (BIGNP (ARRAY A DATA-SEGMENT) (EXP 2 WORD-SIZE))
      (BIGNP (ARRAY B DATA-SEGMENT) (EXP 2 WORD-SIZE))
      (NOT (LESSP MAX-TEMP-STK-SIZE
                  (ADD1 (ADD1 (ADD1 (LENGTH TEMP-STK))))))
      (EQUAL (DEFINITION 'BIG-ADD PROG-SEGMENT) (BIG-ADD-PROGRAM))
      (DEFINEDP A DATA-SEGMENT)
      (DEFINEDP B DATA-SEGMENT)
      (NOT (EQUAL A B))
      (NUMBERP I)
      (LESSP I N)
      (EQUAL N (LENGTH (ARRAY A DATA-SEGMENT)))
      (EQUAL N (LENGTH (ARRAY B DATA-SEGMENT)))
      (BOOLEANP C))
 (EQUAL (P (P-STATE
            '(PC (BIG-ADD . 2))
            (PUSH (P-FRAME
                   (LIST (CONS 'A (TAG 'ADDR (CONS A I)))
                         (CONS 'B (TAG 'ADDR (CONS B I)))
                         (CONS 'N (TAG 'NAT (DIFFERENCE N I))))
                   RET-PC)
                  CTRL-STK)
            (PUSH (TAG 'ADDR (CONS A I))
                  (PUSH (TAG 'BOOL C)
                        TEMP-STK))
            PROG-SEGMENT
            DATA-SEGMENT
            MAX-CTRL-STK-SIZE
            MAX-TEMP-STK-SIZE
            WORD-SIZE
            'RUN)
           (BIG-ADD-LOOP-CLOCK (DIFFERENCE N I)))
        (P-STATE RET-PC
                 CTRL-STK
                 (PUSH (BIG-ADD-CARRY-OUT-LOOP I
                                               (ARRAY A DATA-SEGMENT)
                                               (ARRAY B DATA-SEGMENT)
                                               (DIFFERENCE N I)
                                               (TV C)
                                               (EXP 2 WORD-SIZE))
                       TEMP-STK)
                 PROG-SEGMENT
                 (PUT-ARRAY (BIG-ADD-ARRAY-LOOP I
                                                (ARRAY A DATA-SEGMENT)
                                                (ARRAY B DATA-SEGMENT)
                                                (DIFFERENCE N I)
                                                (TV C)
                                                (EXP 2 WORD-SIZE))
                            A
                            DATA-SEGMENT)
                 MAX-CTRL-STK-SIZE
                 MAX-TEMP-STK-SIZE
                 WORD-SIZE
                 'RUN)))
```

top frame contains bindings for the three locals **A**, **B**, and **N**, and the usual return program counter. The values of the three locals are consistent with the idea that we are considering an arbitrary legal arrival at **LOOP**, not just the first arrival. That is, **A** and **B** point to the **I**th words of their respective data areas and **N** has been decremented by **I** from its original value of **(LENGTH A)**. The temporary stack is also configured as though we were at an arbitrary arrival at **LOOP**. We see two things pushed onto it, a running carry and the current value of **A**. Finally, the instruction count controlling how many instructions we execute is **(BIG-ADD-LOOP-CLOCK (DIFFERENCE N I))**, just enough instructions to take us right through the loop and out via the **RET** instruction.

The final state described by the theorem in Figure 3-5 is similar to that for **BIG-ADD** except that we used the derived functions **BIG-ADD-CARRY-OUT-LOOP** and **BIG-ADD-ARRAY-LOOP**. Furthermore, it is consistent with the idea that we started at an arbitrary arrival at **LOOP**. Thus, the carry flag on the stack is that obtained by starting at position **I** and iterating **N**-**I** times and the final value of **A** is the analogous big number array computed from position **I** onwards.

Because we used **BIG-ADD-CARRY-OUT-LOOP** and **BIG-ADD-ARRAY-LOOP** in this specification it is relatively straightforward to prove this theorem. The induction is on **I** up to **N** by **ADD1**. In the base case, when the difference between **N** and **I** is 1, the proof requires the symbolic execution of 11 Piton instructions (from **LOOP** through the **RET** statement) to reduce the left-hand side to the right-hand side. In the inductive step, the proof requires the symbolic execution of 19 instructions (once around the **LOOP**) to reduce the induction conclusion to the induction hypothesis. No deep (big number specific) problems arise in the proof since **BIG-ADD-CARRY-OUT-LOOP** and **BIG-ADD-ARRAY-LOOP** do exactly the same sequence of **PUT**s as the code. Basically this entire proof is devoted merely to checking that all the preconditions of all the Piton instructions are satisfied under the hypotheses listed and that the step functions compose as claimed by the derived functions. We do not discuss the proof of the theorem further. Instead, we turn to the proof of the correctness of **BIG-ADD** given this theorem about its **LOOP**.

Recall that the correctness of **BIG-ADD** describes an initial state in which we are about to **CALL BIG-ADD** on arguments satisfying the input condition of **BIG-ADD**. We are asked to execute 3+**(BIG-ADD-LOOP-CLOCK N)** instructions. The first instruction is a **CALL** and we build a standard **BIG-ADD** frame with the initial values of **A**, **B**, and **N**. In that frame, **A** and **B** both point to the 0th position of their respective arrays. We then enter the body of **BIG-ADD** and execute two more instructions. These push two things onto the temporary stack, an initial input carry of **F** and the value of **A**. We then arrive at **LOOP** with **(BIG-ADD-LOOP-CLOCK N)** ticks left on the clock. The theorem of Figure 3-5 applies if we let **I** be 0 and **C** be **F**. We therefore conclude that the final state is the corresponding instance of the final p-state of Figure 3-5. Note that this instance is exactly the state described in the correctness of **BIG-ADD** except that in the instance at hand we see the derived functions instead of the original specification functions. That is, the proof would be finished if we just knew that **(BIG-ADD-CARRY-OUT-LOOP 0 A B (LENGTH A) F BASE)** was the same as **(BIG-ADD-CARRY-OUT A B F BASE)** and that **(BIG-ADD-ARRAY-LOOP 0 A B (LENGTH A) F BASE)** was the same as **(BIG-ADD-ARRAY A B F BASE)**.

3.7.3. The Equivalence of the Derived and Original Functions

To complete the proof is remains only to show the equivalence of the derived and original specification functions. We will focus on the **BIG-ADD-ARRAY** versus **BIG-ADD-ARRAY-LOOP** and leave the carry out case for the reader.

The theorem we wish to prove is

Theorem. **BIG-ADD-ARRAY** is **BIG-ADD-ARRAY-LOOP**

```
(IMPLIES (AND (LISTP A)
              (BIGNP A BASE)
              (NUMBERP BASE)
              (LESSP 1 BASE))
         (EQUAL (BIG-ADD-ARRAY A B F BASE)
                (BIG-ADD-ARRAY-LOOP 0 A B (LENGTH A) F BASE))).
```

This formula asserts that if **BIG-ADD-ARRAY-LOOP** iterates **(LENGTH A)** times starting from position **0** the result is the same big number as that produced by **BIG-ADD-ARRAY**.

To prove the theorem above we formulate a more general lemma that can be proved by induction. One hint that this is necessary is that one must inductively unfold **BIG-ADD-ARRAY-LOOP** but when the first argument is **0** it is impossible to have an inductive instance of the theorem with the first argument being one greater. We therefore prove

Lemma.

```
(IMPLIES (AND (NUMBERP I)
              (LESSP I (LENGTH A))
              (BIGNP A BASE)
              (NUMBERP BASE)
              (LESSP 1 BASE))
         (EQUAL (BIG-ADD-ARRAY (NTH-CDR I A)
                               (NTH-CDR I B)
                               C
                               BASE)
                (NTH-CDR I (BIG-ADD-ARRAY-LOOP I A B
                                               (DIFFERENCE (LENGTH A) I)
                                               C BASE)))).
```

The expression **(NTH-CDR i x)** is defined so that it is **(CDR**$^{\mathbf{i}}$ **x)**. The lemma above says that if you use **BIG-ADD-ARRAY** to get the sum of the **I**$^{\text{th}}$ **CDR**s of **A** and **B** you get the same thing as using **BIG-ADD-ARRAY-LOOP** on all of **A** and **B** but starting at the **I**$^{\text{th}}$ position and iterating **(LENGTH A)**-**I** times and then looking at the **I**$^{\text{th}}$ **CDR** of the result. Note that we also generalized the input carry flag from **F** to any **C**. The theorem can be proved by induction on **I** up to **(LENGTH A)** by **ADD1**. The proof is not entirely trivial. For example, we here deal with the question of whether the computation is ‘‘messed up’’ by the repeated ‘‘destruction’’ of **A**. That is, each time **BIG-ADD-ARRAY-LOOP** iterates the **A** it uses subsequently is altered by a **PUT** at the location just inspected. ‘‘Fortunately’’ the alteration occurs to the left of that portion of **A** that the subsequent computation inspects.

The theorem that **BIG-ADD-ARRAY** is **BIG-ADD-ARRAY-LOOP** follows immediately from the lemma by instantiating **I** above with **0** and **C** with **F** and then simplifying **(NTH-CDR 0 x)** to **x** and **(LENGTH A)**-**0** to **(LENGTH A)**.

The corresponding equivalence theorem for the carry out case is similar. Once both of these equivalence theorems are proved the proof of the correctness of **BIG-ADD** follows immediately.

3.8. Summary

We can summarize this chapter as follows. We formally defined big number addition. We showed a Piton program for doing big number addition and built a simple system that called the program. We illustrated Piton states by setting up an addition problem and running it on the Piton machine. We illustrated how Piton programs can be formally specified by theorems in the logic. We showed how specifications in the style proposed can be used, together with symbolic execution, to prove the correctness of Piton programs and systems.

This concludes our discussion of **`BIG-ADD`**. Please recall that this chapter was essentially a detour intended to familiarize the reader with Piton and with how it is possible to specify and prove the correctness of Piton programs from the formal semantics of Piton. However, this report is not about big number arithmetic, how to prove Piton programs correct or even how to program in Piton. It is about how we implemented Piton on the FM8502, how we specified the correctness of that implementation, and how we proved it. Now that the semantics of Piton are clearer, we return to the mainstream of this report.

4. A Sketch of FM8502

We wish to implement Piton on FM8502. In this chapter we explain how FM8502 works and the sense in which it has been implemented correctly in hardware.

The FM8502 is a 32-bit general purpose microprocessor with word addressing. The machine has eight general purpose 32-bit wide registers, numbered 0 through 7. Register 0 serves as the program counter.

The machine has four 1-bit condition code registers, designated ‘‘carry,’’ ‘‘zero,’’ ‘‘overflow,’’ and ‘‘negative.’’

The alu supports the standard operations on 32-bit wide vectors interpreted as twos complement integers, natural numbers and simple bit vectors. It takes as input an opcode, two 32-bit wide operands, and the contents of the four condition code registers and yields as output a 32-bit wide bit vector and four new condition codes.

The instruction word contains a 5-bit *opcode* field, specifying some operation to be performed; an interrupt-enabled bit (currently unused by the processor); a 4-bit *condition code mask*; a 5-bit *operand b* field, logically divided into a 3-bit field specifying a register number and a 2-bit field specifying an address mode; and a 5-bit *operand a* field, logically divided into a 3-bit field specifying a register number and a 2-bit field specifying an address mode.

The FM8502 fetch-execute cycle is as follows:

1. Register 0 is used as the program counter. Its contents is treated as a memory address. The value of that address is fetched and decoded into an opcode, condition code mask, and operands b and a, as described above. Register 0 is incremented by one.
2. The pre-decrement computation (defined below) is performed on operand a.
3. The effective address (defined below), e_2, specified by a is determined;
4. The contents, x_2, of e_2 is fetched;
5. The pre-decrement computation is performed on b;
6. The effective address, e_1, specified by b is determined;
7. The contents, x_1, of e_1, is fetched;
8. The operation indicated by opcode is performed by the alu on x_1 and x_2 and the contents of the four condition code registers, yielding an output bit vector, y, and four condition codes;
9. The four condition codes are stored into their respective condition code registers provided the corresponding bit in the condition code mask is set;
10. The output bit vector, y, is stored into address e_1;
11. The post-increment computation (defined below) is performed on a.
12. The post-increment computation is performed on b.

The four address modes are: register, register-indirect, register-indirect with pre-decrement, and register-indirect with post-increment.

The *pre-decrement computation* on a register number and an address mode decrements the contents of the register if the address mode is ‘‘register-indirect with pre-decrement’’ and does nothing otherwise.

The *effective address* specified by a register number and an address mode is the indicated register if the address mode is ''register'' and is the contents of the indicated register otherwise.

The *post-increment computation* on a register number and an address mode increments the contents of the register if the address mode is ''register-indirect with post-increment'' and does nothing otherwise.

The opcodes can be divided into two groups: arithmetic/logical opcodes and conditional move opcodes. The arithmetic/logical opcodes are unconditional move, increment, add with carry, add, negate, decrement, subtract with borrow, subtract, rotate right through carry, arithmetic shift right, logical shift right, exclusive or, or, and, and not. The eight conditional move opcodes permit the second operand to be moved into the address specified by the first conditionally on the setting of any single condition code register. A jump or conditional jump can be coded as a move or conditional move into register 0. Left shifting can be performed by adding a quantity to itself.

It should be noted that the only quantities stored in the FM8502's memory and registers are 32-bit wide bit vectors. The FM8502 alu operates on such bit vectors. The machine does not contain integers or natural numbers, despite the fact that it has instructions with mnemonics like ''add'' and ''subtract.'' However, Hunt proves that the bit vectors produced by the alu are correct with respect to the various expected interpretations of the input vectors. For example, the output of the add instruction, if interpreted as a natural number, is the natural number sum of the natural interpretations (in binary notation) of the input, with suitable accounting for the input and output carry flags. However, the same output can also be interpreted as the integer sum of the integer interpretation of the inputs (in twos complement notation), with suitable accounting for the input carry and the output overflow and negative flags.

FM8502 is formalized in the function **`FM8502`** which is defined on page 163. The state of the FM8502 machine is a six-tuple called an *m-state* which contains the register file, the four condition code bits, and the memory. The function **`FM8502`** takes two arguments, an m-state and a natural number n and returns the m-state obtained by executing n machine instructions. FM8502 does not halt or cause errors—every bit vector in its memory can be interpreted as a legitimate instruction.

While it is not important to Piton, the reader may well wonder in what sense FM8502 is verified. FM8502 is verified in exactly the same sense that FM8501 is verified in Hunt's dissertation [11]. We will briefly discuss Hunt's FM8501 work and then return to FM8502.

Hunt described FM8501 by defining a function called **`SOFT`** (for ''software''). **`SOFT`** is an interpreter for the machine code of a 16-bit wide general purpose computer. The function takes as its arguments a collection of 16-bit wide bit vectors denoting the values of some ''registers,'' ''flags,'' and ''memory,'' and a list called the *oracle* which determines how many instructions are to be executed. **`SOFT`** delivers as its answer a corresponding collection of bit vectors produced after executing the specified number of machine code instructions fetched from the memory.

Hunt then implemented **`SOFT`** at the gate level by exhibiting another function, named **`BIG-MACHINE`**, which is a formal register-transfer model. **`BIG-MACHINE`** has 20 arguments. Fourteen are either fixed length bit vectors or fixed-length lists of fixed-length bit vectors; these arguments, called the *internal registers*, represent the internal state-holding components of the FM8502 implementation (e.g., the register file as well as ''hidden'' resources such as the microcode address register). Three more arguments are also either fixed-length bit vectors or fixed-length lists of fixed-length bit vectors and are called *input latch registers*. The last three arguments of **`BIG-MACHINE`** are formal models of *external devices* such as the sequence of inputs that impinge upon FM8501's data acknowledgment input latch register during a given interval of time.

Roughly speaking, **BIG-MACHINE** recurses once for each discrete time point in the given interval. On each iteration, the external devices are used to determine the new values of the input latch registers; the old values of the input latch registers and the internal registers are used to compute the new values of internal registers. Furthermore, the new values of the internal registers are determined entirely by combinational logic expressions.[9]

Reflection shows that **BIG-MACHINE** can be built as follows: Allocate a physical register (of the appropriate size) to each internal register and input latch. Wire these registers with the combinational logic shown by Hunt so that on each clock pulse the new value of each internal register is computed and stored back into the appropriate register. Wire the input latches so that on each clock pulse they are written by the input lines to FM8501. Thus, each clock pulse causes the hardware to step forward once in the recursion of **BIG-MACHINE**.

Hunt then proved that any computation carried out by **SOFT** can be carried out on **BIG-MACHINE**. In particular, he proved that the state returned by **SOFT** by stepping forward n steps from some initial state may be alternatively computed by mapping the state down to an initial state of **BIG-MACHINE**, stepping that machine forward some k steps, and then projecting out of **BIG-MACHINE**'s state those six components that represent a **SOFT** state.

The number of steps, k, taken by **BIG-MACHINE**, is a function of the oracle used in **SOFT**. Because **BIG-MACHINE** is microcoded, a single instruction at the **SOFT** level takes many instructions at the **BIG-MACHINE** level. Hunt defined a constructive function that determines how many microcycles **BIG-MACHINE** takes to carry out a computation performed by **SOFT** from a given initial state and oracle. The situation is more complicated than one might at first expect because the microcode machine must enter wait states while waiting, for example, for the memory to respond to read requests. In order to say how many microcycles elapse it is necessary to say how much time is spent in each such wait state—a question that cannot be answered from the model Hunt develops since, quite rightly, the external devices are not implemented in his model. The role of the oracle in **SOFT** is to specify, for each instruction executed, how many cycles are spent in each of the possible wait states. This is why the oracle is a list of length n rather than simply n. The actual choices for the lengths of the wait states are unimportant to the final answer delivered, a fact easily proved from the observation that the definition of **SOFT** makes no reference to the **CAR** of the oracle, it only **CDR**s it until it is no longer a **LISTP**.

In summary, Hunt defined two machines, **SOFT** and a register transfer model, and proved them equivalent. ''FM8501'' (which is a term not formally defined in [11]) can be thought of as the machine these two functions describe.

''FM8502'' has been produced in a similar fashion. In particular, there is a 32-bit wide version of **SOFT** describing the machine used here and a corresponding register transfer model that has been proved equiv-

---

[9]This summary of Hunt's work completely ignores what is perhaps the key contribution of his dissertation. The combinational logic used in the FM8501 specification is actually generated from recursive functions that operate on bit vectors of arbitrary word size. Proving such logic correct was inductive. Thus Hunt was able to avoid the combinatoric explosion usually associated with proving the correctness of large logic expressions. The gate graph for FM8501 is obtained after the proof of correctness, by instantiating the word size to 16 and unfolding the recursive functions. If one ignores our change in the handling of condition codes, FM8502 is obtained by instantiating the same functions to word size 32.

alent. The 32-bit wide version of **SOFT**, its gate-level description and the equivalence proof were constructed by Warren Hunt. The 32-bit wide **SOFT** is formally defined on page 167. We do not include the register transfer model of **SOFT** in this report. For our purposes it was inconvenient that **SOFT** takes seven arguments (six representing the input state and one representing the oracle) and returned a list of six items (representing the output state). We therefore defined the symbol **FM8502** to be a packaging of **SOFT** that takes an initial m-state and a number of steps and returns a final m-state. See Chapter 8.

5. The Correctness of Piton on FM8502

The implementation of Piton on FM8502 is via what might be called a ‘‘cross-loader’’ written as a function, **LOAD**, in the logic. The function takes a Piton state and generates an FM8502 state or ‘‘core image.’’

The correctness theorem is a formalization of the following classic commutative diagram:

$$\begin{tikzcd}[column sep=huge, row sep=huge] p_0 \arrow[r, dashed, "\mathbf{n} \text{ Piton steps}"] \arrow[d, "\mathbf{LOAD}"'] & p_n \arrow[d, "\mathbf{DISPLAY\text{-}M\text{-}DATA\text{-}SEGMENT}"] \\ m_0 \arrow[r, dashed, "\mathbf{k} \text{ FM8502 steps}"'] & m_k \end{tikzcd}$$

The basic idea is that one has some initial Piton state, p_0, and one wishes to obtain the state produced by running Piton forward **n** steps. However, depending on one’s point of view, the abstract Piton machine does not ‘‘really’’ exist or is too expensive or inefficient to use. The correctness theorem tells us there is another way: map the Piton state ‘‘down’’ to an FM8502 state, run FM8502, and then map the resulting state back ‘‘up.’’ However, the situation is much more subtle than suggested by the diagram.

The correctness theorem may be paraphrased as follows. Suppose p_0 is some proper p-state that is loadable on FM8502 and has word size 32. Let p_n be the p-state obtained by running the Piton machine **n** steps on p_0. Suppose p_n is not erroneous. Suppose furthermore that the ‘‘type specification’’ (see below) for the data segment of p_n is **ts**. Then it is possible to obtain the data segment of p_n via FM8502 as follows:

- *Down*. Let the initial FM8502 state, m_0, be **(LOAD** p_0**)**.
- *Across*. Obtain a final FM8502 state, m_k, by running FM8502 **k** steps on m_0, where **k** is a number obtained from p_0 and **n** by the constructive function **FM8502-CLOCK**.
- *Up*. Apply the constructive function **DISPLAY-M-DATA-SEGMENT** to (a) the final FM8502 state, m_k, (b) the final type specification, **ts**, and (c) the link tables (computed by the constructive function **LINK-TABLES** from p_0).

This is formalized as

Theorem. FM8502 Piton is Correct

```
(IMPLIES (AND (PROPER-P-STATEP P0)
              (P-LOADABLEP P0)
              (EQUAL (P-WORD-SIZE P0) 32)
              (EQUAL PN (P P0 N))
              (NOT (ERRORP (P-PSW PN)))
              (EQUAL TS (TYPE-SPECIFICATION (P-DATA-SEGMENT PN))))
         (EQUAL (P-DATA-SEGMENT PN)
                (DISPLAY-M-DATA-SEGMENT
                  (FM8502 (LOAD P0) (FM8502-CLOCK P0 N))
                  TS
                  (LINK-TABLES P0)))).
```

We believe this theorem is merely a formalization of what is usually meant by the informal remark that a programming language is implemented correctly on given hardware base. Nevertheless (or perhaps, *because* it is an accurate formalization), as a correctness result the theorem is very subtle. What does this statement tell the user of FM8502 Piton? The rest of this chapter is devoted to a discussion of the meaning of this theorem. We discuss each of the hypotheses of the theorem and then what the conclusion tells us. We then apply the theorem to the **BIG-ADD** program of Chapter 3 to answer the question ‘‘Can we use FM8502 to add big numbers?’’ During that discussion we reconsider each of the hypotheses and the conclusion again.

5.1. The Hypotheses of the Correctness Result

5.1.1. Proper P-States

In the first place, we are interested only in proper p-states. Those are the p-states in which all components are syntactically well-formed and compatible.

We could have chosen to specialize the correctness theorem still further and require that **P0** be an ‘‘initial’’ state. Without loss of computing power, we could define an initial state to be one in which the temporary stack is empty and we are about to execute the first instruction in the subroutine named **MAIN**, (which has no local variables). We did not do this only because the proof requires a more general treatment of the mapping down from Piton to FM8502 and having paid for it we decided to provide it to the user.

5.1.2. Loadable

P0 must be ‘‘loadable’’ in the sense that the total size of the compiled programs, stacks, and data must not exceed the memory capacity of the FM8502. Thus, the FM8502 implementation of Piton is correct only for a subset of the abstract Piton state space.

It should be understood that the determination of whether a particular state is loadable is dependent not just on the resource limitations declared by the user (in the form of the maximum stack sizes) but also on the amount of code generated by our compiler. This immediately contaminates the hypothesis of the correctness theorem with implementation details. Ideally one would like the correctness theorem for a programming language implementation to read something like ‘‘if the abstract state satisfies these conditions (all expressed in familiar abstract terms), then the computation can be carried out on the concrete machine as follows ...’’ But here we have a condition on the abstract state that, if one delves into its formal definition, involves such implementation details as how many words of machine code are generated for the **PUSH-CONSTANT** instruction. We find this unaesthetic but unavoidable. The abstract Piton machine has no inherent resource limitations—the user is free to specify any desired stack sizes. The concrete FM8502, on the other hand, is fundamentally limited to at most 2^{32} words of memory. How quickly those finite resources are exhausted depends on how they are used by the implementation.

5.1.3. Word Size 32

Recall that the definition of Piton is parameterized by the word size. The word size is an explicit part of the Piton state and the abstract Piton machine is sensibly defined for all word sizes. However, the correctness theorem hypothesizes that the word size is 32. That is, the FM8502 implementation of Piton is correct only for word size 32.

This reflects the fact that the FM8502 is a 32-bit wide machine. Of course, our implementation could have allotted two FM8502 words to each Piton object and thus implemented word size 64. Indeed, the implementation could have been a nontrivial function of the word size and allotted as many FM8502 words as necessary to accommodate the user's declared word size, with the concomitant complications in the implementation of arithmetic and all other operations. We mention these possibilities only to alert the reader to the fact that the FM8502 implementation of Piton is *unnecessarily* restricted to a slice of the abstract state space. Nevertheless, such restrictions are standard practice in language implementations.

5.1.4. Non-Erroneous Final State

The theorem next assumes that the final Piton state is non-erroneous. That is, were this program run on the abstract Piton machine, the final psw would be either **RUN** or **HALT**. But we are imagining that the program was run on FM8502, not the abstract Piton machine. Can we tell by looking at the FM8502 core image whether the abstract machine would have caused an error? No. The only way this hypothesis can be relieved is for the user to have proved that (a) his program runs without error on data satisfying its input conditions and (b) the data used in this particular run satisfies those conditions.

5.1.5. Knowledge of the Final Type Specification

The next assumption is that **TS** is the type specification of the final Piton state. We have not discussed type specifications before. A *type specification* for a data segment is a structure isomorphic to the data segment except that where the data segment has a data object the type specification has just the type of the object. For example, the type specification for the data segment

```
((LEN (NAT 5))
 (A   (NAT 0) (NAT 1) (NAT 2) (NAT 3) (NAT 4))
 (X   (INT -23)
      (NAT 7)
      (BOOL T)
      (BITV (1 0 1 0 1 1 0 0))
      (ADDR (A . 3))
      (PC (SETUP . 25))
      (SUBR MAIN)))
```

is

```
((LEN NAT)
 (A   NAT NAT NAT NAT NAT)
 (X   INT
      NAT
      BOOL
      BITV
      ADDR
      PC
      SUBR)).
```

In order to reconstruct the data segment of the final Piton state, the user must *know* the type specification of that state. ‘‘Whoa! Do you mean the programmer has to know the type of every location in his final Piton data segment to read the answers from the FM8502 core image?’’ Yes, he does. ‘‘The FM8502 core image doesn't tell him what the types of the objects are?’’ No, it doesn't. ‘‘Then if he doesn't have an abstract Piton machine, how can he know what the final types are?’’ Proof.

This is not a new idea, but it looks a little startling when expressed formally. When the assembly language programmer inspects word 27,349 of his final machine state and sees

```
B11111111111011010010100101111001
```

he knows that it might be the integer -1,234,567 in twos complement notation, or the natural 4,293,732,729 in binary notation, or a bit mask, or the address of some subroutine or data word. He knows how to interpret it because he understands his program.

The idea of **TS** is startling because it is defined in terms of **PN**, the supposedly unknown final state. A naive view of the correctness theorem suggests the following paradoxical reading: to determine the final data segment the programmer must obtain the final data segment, get its type specification, and then inspect the core image. This is not the recommended approach!

How can one know the final type specification without obtaining the final data segment? One can prove that one's interpretation of the final state is correct. For a Piton program to be useful via the FM8502 implementation, one must not only prove that it does not cause errors when called as expected but that it delivers a final data segment with the expected type specification. This is not an onerous task. It is usually part of the specification anyway. We illustrate this when we apply the correctness result to **BIG-ADD** later in this chapter (page 60).

If Piton had a strongly typed syntax—so that it was impossible to change the type of a variable or data location—then the final type specification would be the same as the initial one and the theorem would no longer even suggest that **PN** is needed to recover the data segment of **PN**. This suggests another approach using FM8502 Piton: The Piton programmer could constrain himself to those Piton programs that do not change their type specifications and prove that he has done so.

5.2. The Conclusion of the Correctness Result

The above discussion may be summarized as follows: The Piton programmer must know that his program is proper, that the static size of the FM8502 image is not excessive, and that the word size is 32. He must know that the final state is non-erroneous on this execution and he must know the final type specification of the data segment. What does the conclusion tell him?

5.2.1. The Final Data Segment

First of all, it does not permit him to reconstruct the entire final state, only its data segment. For example, we do not say how to recover from the final FM8502 state the program segment of the final p-state. This is impossible (without some additional information) since, for example, the symbolic names of the Piton programs, data areas, variables, and labels are all discarded by **LOAD**. Furthermore, it is pointless to reconstruct the program segment: it never changes anyway. More questionable, perhaps, is that we do not show how to recover the final control or temporary stacks.

We felt it was sufficient to handle the data segment alone. If the programmer has a program whose final answer is left on one of the stacks, he could add the additional code that moves that answer into the data area, simply to get total reassurance that what he sees in FM8502's final state is correct. That is what we did in the **MAIN** program (page 29): after calling **BIG-ADD** we pop the ''carry out'' flag off the temporary stack and store it into the global variable **C**. Thus, the implementation correctness theorem enables us to get both of **BIG-ADD**'s answers (the final array and the flag) out of the **FM8502** core image.

5.2.2. The FM8502 Route

If all of the hypotheses are true for a particular run and the user is happy to see just the final data segment, this theorem tells him how to get it from FM8502 and the initial state, **P0** and **N**. In particular, it is

```
(DISPLAY-M-DATA-SEGMENT (FM8502 (LOAD P0)
                                (FM8502-CLOCK P0 N))
                        TS
                        (LINK-TABLES P0)).
```

The functions, **LOAD**, **LINK-TABLES**, and **DISPLAY-M-DATA-SEGMENT** are all defined in the formal treatment of the correctness result. **FM8502-CLOCK** is discussed below and on page 205.

Informally, **LOAD** constructs an FM8502 binary core image from a Piton state by compiling, assembling and linking the Piton programs and data areas. **FM8502-CLOCK** determines how many clock ticks it takes FM8502 to carry out the computation performed by Piton in **N** ticks—a determination made by carrying out the computation with the abstract Piton machine and counting how many FM8502 instructions are used in our implementation. This may strike some readers as problematic and we will return to it below. **LINK-TABLES** is part of the **LOAD** function. It takes a Piton state and computes several tables that tell the linker where each program, label, stack, and data area is to be located in absolute memory space.

DISPLAY-M-DATA-SEGMENT reconstructs the Piton data segment by scanning the type specification **TS** to determine the names of the data areas, using the link tables to determine the absolute location at which each data area was allocated, scanning the memory of the final state from those locations to recover bit vectors, and finally using the type specification again to convert the individual bit vectors back into Piton data objects of the appropriate type. The formalization of this concept is embodied in the function **DISPLAY-M-DATA-SEGMENT** which is defined on page 207.

We suggest that this is exactly what the assembly language programmer (or debugger) does when inspecting a core dump.

5.3. The Termination of FM8502

Our handling of the termination of Piton programs on FM8502 leaves something to be desired. As things stand now, the correctness theorem tells you that if you inspect the FM8502 state on the $\mathbf{k}^{th}$ clock tick, where **k** is **(FM8502-CLOCK P0 N)**, then it ‘‘maps up’’ as described. But the theorem says nothing about what FM8502 does on the $\mathbf{k}+1^{st}$ clock tick. Furthermore, the only way to determine **k** is to use **FM8502-CLOCK**, which runs the computation on the abstract Piton machine.

The preferred way to read the correctness theorem is that it tells us ‘‘there exists a number **k** such that if you run FM8502 **k** ticks you get a suitable state.’’ **FM8502-CLOCK** is a ‘‘witness function’’ that exhibits a suitable **k**. Its definition is very dependent upon our implementation and proof. We do not exhibit a definition of **FM8502-CLOCK** in this report, though we discuss it again on page 205.

If FM8502 were built as it now stands and the Piton implementation were left as is, it could still provide a practical means of carrying out computation: The user could write his top-level Piton program to set the global variable **TERMINATED** to true and enter an infinite loop when the actual computation has completed. Then, he should prove that when **TERMINATED** is true the answer is correct. He can then execute his program on FM8502 and periodically map up to see if **TERMINATED** is true.

The main reason that the issue of the clock has not been more conveniently addressed in our correctness theorem is that FM8502 does not yet actually exist. When it does, there will be some kind of monitor to which Piton returns when complete, and some kind of output lines upon which either Piton or FM8502 will signal completion. When we have a practical means of interfacing FM8502 to the world, we will implement a suitable reflection of it for Piton and modify the correctness theorem appropriately.

5.4. Applying the Correctness Result to BIG-ADD

Recall that in Chapter 3 we defined a small system of two Piton programs for doing big number addition. Suppose we are now interested in running that system on the FM8502 implementation of Piton. What does the correctness result tell us?

The system in question is constructed by the function

Definition.
```
(SYSTEM-INITIAL-STATE A B)
   =
(P-STATE '(PC (MAIN . 0))
         '((NIL (PC (MAIN . 0))))
         NIL
         (LIST (MAIN-PROGRAM)
               (BIG-ADD-PROGRAM))
         (LIST (CONS 'BNA A)
               (CONS 'BNB B)
               (CONS 'N (LIST (TAG 'NAT (LENGTH A))))
               (CONS 'C (LIST (TAG 'NAT 0))))
         10
         8
         32
         'RUN).
```

We defined the acceptable values of **A** and **B** with

Definition.
```
(SYSTEM-INITIAL-STATE-OKP A B)
   =
(AND (BIGNP A (EXP 2 32))
     (BIGNP B (EXP 2 32))
     (EQUAL (LENGTH A) (LENGTH B))
     (NOT (ZEROP (LENGTH A)))
     (LESSP (LENGTH A) (EXP 2 32))).
```

We proved the following theorem about this Piton system:

Theorem. Correctness of a **BIG-ADD** System

```
(IMPLIES
 (SYSTEM-INITIAL-STATE-OKP A B)
 (EQUAL (P (SYSTEM-INITIAL-STATE A B)
           (SYSTEM-INITIAL-STATE-CLOCK A B))
        (P-STATE
         '(PC (MAIN . 5))
         '((NIL (PC (MAIN . 0))))
         NIL
         (LIST (MAIN-PROGRAM)
               (BIG-ADD-PROGRAM))
         (LIST (CONS 'BNA (BIG-ADD-ARRAY A B F (EXP 2 32)))
               (CONS 'BNB B)
               (CONS 'N (LIST (TAG 'NAT (LENGTH A))))
               (CONS 'C (LIST (BIG-ADD-CARRY-OUT A B F (EXP 2 32)))))
         10 8 32 'HALT))).
```

Suppose we have in mind some particular big numbers **a** and **b** such that **(SYSTEM-INITIAL-STATE-OKP a b)** holds. We know how to add them together with the abstract Piton machine: let $\mathbf{p_0}$ be **(SYSTEM-INITIAL-STATE a b)** and let $\mathbf{p_n}$ be the result of running the Piton machine **(SYSTEM-INITIAL-STATE-CLOCK a b)** steps starting from $\mathbf{p_0}$. Then by the correctness of the **BIG-ADD** system (above) we know that the data segment of $\mathbf{p_n}$ contains the correct big number sum. Can we get that same answer via FM8502?

It is not at all obvious that FM8502 Piton can do the job for us, given the complexity of our theorem stating the correctness of FM8502 Piton. We must know five things. First, we must know that $\mathbf{p_0}$ is a proper p-state. Second, we must know that $\mathbf{p_0}$ is loadable. Third, we must know that the word size of $\mathbf{p_0}$ is 32. Fourth, we must know that the final state, $\mathbf{p_n}$, is non-erroneous. Fifth, we must know the type specification of the data segment of $\mathbf{p_n}$. We deal with each of these issues in turn. Recall that $\mathbf{p_0}$ is **(SYSTEM-INITIAL-STATE a b)** and we know **(SYSTEM-INITIAL-STATE-OKP a b)**.

5.4.1. Proper P-States

We can prove that $\mathbf{p_0}$ is proper from the assumption that **a** and **b** satisfy **SYSTEM-INITIAL-STATE-OKP**. In particular,

Theorem.

```
(IMPLIES (SYSTEM-INITIAL-STATE-OKP A B)
         (PROPER-P-STATEP (SYSTEM-INITIAL-STATE A B))).
```

This theorem is straightforward to prove. Recall that **PROPER-P-STATEP** is concerned with such issues as whether the programs in the program segment are syntactically well-formed and mention no global data other than that declared in the data segment and that all of the Piton objects involved in the state are legal. But the program segment in this case is a constant, namely the list containing the **MAIN** program and **BIG-ADD**, and the data segment is ‘‘almost’’ a constant—the names of the areas are explicitly given and their contents are derived from **A** and **B**. So it is easy to confirm that our programs are proper and only refer to declared data. It is interesting that this is the first place in the **BIG-ADD** exercise where we are concerned with the question of whether our programs are syntactically well-formed. This is because the abstract Piton semantics gives some meaning even to ill-formed programs and we proved that **BIG-ADD** was correct with respect to that semantics. Only when we contemplate compiling it do we need to make sure the program is well-formed, since there is no guarantee in our implementation correctness theorem that the compiler agrees with the abstract semantics on ill-formed programs. In addition to the syntactic checks discussed above, **PROPER-P-STATEP** checks that all Piton objects in the initial state are legal. The proof that this is true for **(SYSTEM-INITIAL-STATE A B)** appeals to the **SYSTEM-INITIAL-STATE-OKP** hypothesis, which guarantees that **A** and **B** are big numbers, and hence the **BNA** and **BNB** data areas contain legal Piton natural numbers.

5.4.2. Loadable

The next question is whether $\mathbf{p_0}$ is loadable. That depends on how big **A** and **B** are. If each is of length 2^{100}, $\mathbf{p_0}$ is certainly not loadable since its data segment contains two areas each of which require 2^{100} words. On the other hand, for ‘‘small’’ **A** and **B** the system initial state is loadable. We have proved mechanically

Theorem.
```
(IMPLIES (AND (SYSTEM-INITIAL-STATE-OKP A B)
              (LESSP (LENGTH A) 1000))
         (P-LOADABLEP (SYSTEM-INITIAL-STATE A B))),
```

which guarantees that if **a** has fewer than a thousand digits then $\mathbf{p_0}$ is loadable. We could have proved a much higher bound on the size of **A** and **B** but we felt that this bound makes the point that we can do significantly large additions with this system.

Recall that **P-LOADABLE** is implementation dependent. Thus, the proof of the above theorem actually involves reasoning about the data representation and the compiler. We sketch the proof here simply to reassure the reader that this ‘‘implementation dependent’’ reasoning does not require undue familiarity with the FM8502 implementation of Piton. The program segment created by **SYSTEM-INITIAL-STATE** is constant and so its FM8502 size is constant and may be determined by computation to be 97. (That is, if you call the compiler on **MAIN** and **BIG-ADD** and add the lengths of the resulting programs you get 97.) The space allocated to the Piton stacks in this example is also constant and is seen to be 23. The data segment constructed by **SYSTEM-INITIAL-STATE** has four areas, one each for **A** and **B** (each containing **(LENGTH A)** words) and a word each for **N** and **C**. Thus the total size of the loaded system is 97+23+2+2n, where n is **(LENGTH A)**. The theorem has thus been reduced to showing that if n<1000 then $122+2n < 2^{32}$. This is trivial.

5.4.3. Word Size 32

The third issue is the word size of $\mathbf{p_0}$. It is easy to prove

Theorem.
```
(EQUAL (P-WORD-SIZE (SYSTEM-INITIAL-STATE A B)) 32).
```

5.4.4. Non-Erroneous Final State

The fourth issue is establishing that the final state, $\mathbf{p_n}$, is non-erroneous. Recall that $\mathbf{p_n}$ is

```
(P (SYSTEM-INITIAL-STATE a b)
   (SYSTEM-INITIAL-STATE-CLOCK a b)).
```

However, from the correctness of the **BIG-ADD** system we know that the final psw of this state is in fact **HALT** and so the state is non-erroneous. More generally we can prove mechanically

Theorem.

```
(IMPLIES (SYSTEM-INITIAL-STATE-OKP A B)
         (NOT (ERRORP (P-PSW (P (SYSTEM-INITIAL-STATE A B)
                                (SYSTEM-INITIAL-STATE-CLOCK A B)))))).
```

Thus, the fact that the FM8502 implementation of Piton is correct only for non-erroneous computations is not a problem because we are using it to run verified code and we proved that our code does not cause errors when run on input satisfying our input conditions.

5.4.5. Knowledge of the Final Type Specification

The fifth issue is knowledge of the type specification of the final data segment. This was perhaps the most problematic aspect of our implementation correctness theorem. Recall that the problem is that to recover the final data segment from the final FM8502 core image it is necessary to *know* the types of the objects in the final data segment. But it is clearly impractical to obtain that knowledge by computing the final data segment with the abstract Piton semantics since there would then be no need to compute with FM8502. We argued that the final type specification could be obtained via proof.

The following mechanically proved theorem illustrates this argument.

Theorem.

```
(IMPLIES (SYSTEM-INITIAL-STATE-OKP A B)
         (EQUAL (TYPE-SPECIFICATION
                 (P-DATA-SEGMENT
                  (P (SYSTEM-INITIAL-STATE A B)
                     (SYSTEM-INITIAL-STATE-CLOCK A B))))
                (LIST (CONS 'BNA (FOR X IN A COLLECT 'NAT))
                      (CONS 'BNB (FOR X IN B COLLECT 'NAT))
                      (CONS 'N   (LIST 'NAT))
                      (CONS 'C   (LIST 'BOOL))))).
```

This theorem tells us that for **A** and **B** satisfying **SYSTEM-INITIAL-STATE-OKP** the type specification of the data segment of the final abstract p-state is

```
(LIST (CONS 'BNA (FOR X IN A COLLECT 'NAT))
      (CONS 'BNB (FOR X IN B COLLECT 'NAT))
      (CONS 'N   (LIST 'NAT))
      (CONS 'C   (LIST 'BOOL))).
```

That is, the final value of the **BNA** array is a list of natural numbers. The final value of the **BNB** array is also a list of natural numbers. The final value of **N** is a natural. The final value of **C** is a Boolean. Note that **C** changed type in the computation. Its initial value was a natural. This theorem is easy to prove given the correctness of the **BIG-ADD** system. After all, the final value of **BNA** is a big number and so is a list of naturals, the final value of **C** is the carry out, which is a Boolean, and **BNB** and **N** are unchanged by the computation. Since the system is correct, we know the final type specification even without knowing the final data segment.

5.4.6. Using FM8502 to Add Big Numbers

Given all of the foregoing we know that if **A** and **B** satisfy **SYSTEM-INITIAL-STATE-OKP** then FM8502 can be used to add them together. Another way to view the situation is that we can combine the correctness of the **BIG-ADD** system with the correctness of the FM8502 implementation of Piton and eliminate entirely the semantics of Piton.

The theorem is

Theorem.
```
(IMPLIES
 (AND (SYSTEM-INITIAL-STATE-OKP A B)
      (LESSP (LENGTH A) 1000))
 (EQUAL (DISPLAY-M-DATA-SEGMENT
              (FM8502 (LOAD (SYSTEM-INITIAL-STATE A B))
                      (FM8502-CLOCK (SYSTEM-INITIAL-STATE A B)
                                    (SYSTEM-INITIAL-STATE-CLOCK A B)))
              (LIST (CONS 'BNA (FOR X IN A COLLECT 'NAT))
                    (CONS 'BNB (FOR X IN B COLLECT 'NAT))
                    (CONS 'N   (LIST 'NAT))
                    (CONS 'C   (LIST 'BOOL)))
              (LINK-TABLES (SYSTEM-INITIAL-STATE A B)))
        (LIST (CONS 'BNA (BIG-ADD-ARRAY A B F (EXP 2 32)))
              (CONS 'BNB B)
              (CONS 'N (LIST (TAG 'NAT (LENGTH A))))
              (CONS 'C (LIST (BIG-ADD-CARRY-OUT A B F (EXP 2 32))))))).
```

Observe that the theorem does not mention **P**, proper p-states, erroneous computations or type specifications. It says that if you have acceptable big numbers **A** and **B** then you can add them together by creating an FM8502 state, running FM8502, and then applying **DISPLAY-M-DATA-SEGMENT** to the final state.

As stated above the theorem is slightly unsatisfying because the second and third arguments to **DISPLAY-M-DATA-SEGMENT** still involve **A** and **B**. In fact, the expressions in those argument positions are functions only of the the length of **A**. That is, given the length of **A** we can construct the type specification and the link tables without any further knowledge of **A** or **B**. We can thus define the function **DISPLAY-ANSWERS** so that it takes only the final m-state and the length of the big number system and returns the list consisting of the sum and carry out. **DISPLAY-ANSWERS** has built into it the types and absolute locations of the answers in the final core image. We can then repackage the above into

Theorem.
```
(IMPLIES (AND (SYSTEM-INITIAL-STATE-OKP A B)
              (LESSP (LENGTH A) 1000))
         (EQUAL
          (DISPLAY-ANSWERS
           (FM8502 (LOAD (SYSTEM-INITIAL-STATE A B))
                   (FM8502-CLOCK (SYSTEM-INITIAL-STATE A B)
                                 (SYSTEM-INITIAL-STATE-CLOCK A B)))
           (LENGTH A))
          (LIST (BIG-ADD-ARRAY A B F (EXP 2 32))
                (BIG-ADD-CARRY-OUT A B F (EXP 2 32))))).
```

The above theorems demonstrating that FM8502 can do big number arithmetic would be hard to prove in the absence of our results on Piton, of course. Indeed, the beauty of the current arrangment of theorems is that the difficult, problem specific reasoning that establishes our programs correct is done with respect to the elegant abstract semantics of Piton. But our programs can be efficiently carried out by the concrete FM8502.

It is often difficult to ascertain whether a given formal sentence adequately captures the informal notions offered in explanation. Does our correctness result for the FM8502 implementation of Piton actually capture the alleged idea? Without a formal ‘‘theory of implementations’’ that question cannot be answered. However, we can ‘‘stack’’ the correctness of a given Piton program on top of the correctness of

FM8502 Piton to get a theorem that eliminates the intermediate level, i.e., Piton. We offer this as evidence that our correctness theorems are adequate.

5.4.7. Concrete Data

We now return to the question that started our analysis of **BIG-ADD** versus the correctness result. Imagine that we have two particular big numbers we wish to add, say **'((NAT 246838082) (NAT 3116233281) (NAT 42632655) (NAT 0))** and **'((NAT 3579363592) (NAT 3979696680) (NAT 7693250) (NAT 0))**. Can we use FM8502 to add them? We now know the answer is ''yes'' provided the two numbers satisfy **SYSTEM-INITIAL-STATE-OKP** and the numbers have fewer than one thousand digits. It is a theorem that

Theorem.
```
(AND (SYSTEM-INITIAL-STATE-OKP '((NAT 246838082)
                                 (NAT 3116233281)
                                 (NAT 42632655)
                                 (NAT 0))
                               '((NAT 3579363592)
                                 (NAT 3979696680)
                                 (NAT 7693250)
                                 (NAT 0)))
     (LESSP (LENGTH '((NAT 246838082)
                      (NAT 3116233281)
                      (NAT 42632655)
                      (NAT 0)))
            1000)).
```

(Observe that this theorem is trivial to prove by computation.) Thus, if we run FM8502 on the corresponding initial state and extract the answers they will be correct. The point is that despite all our analysis and proofs, we know that the final answer is correctly only if we know we are using the program in accordance with its specifications.

One last point deserves to be made here. Since we did not design the big number addition system to signal its own termination we must know, for each concrete input data set, how long to let the machine run. For big numbers **A** and **B**, the FM8502 must run for

```
     (FM8502-CLOCK (SYSTEM-INITIAL-STATE A B)
                   (SYSTEM-INITIAL-STATE-CLOCK A B))
```

instructions. For the concrete data above, this expression is equal to 190.

6. The Implementation of Piton on FM8502

To implement Piton on FM8502 we must define **LOAD**. **LOAD** takes a p-state as its input and produces an m-state as its output. **LOAD** is defined on page 198.

We implement **LOAD** in three steps.

- The first is the *resource representation* phase. In this step we are concerned with using the resources of the FM8502 to represent the stacks and other resources of the Piton machine. The principal output of this phase is a symbolic description of the contents of the FM8502 registers and that portion of the memory containing the stacks. This symbolic description will be subjected to ‘‘linking’’ (below). The formalization of this concept is embodied in the function **P->R** which is defined on page 199.
- Second, we translate Piton instructions into FM8502 instructions. We call this *compiling* the Piton programs. In our implementation, the compiler does not produce binary machine code; instead it produces what we call ‘‘i-code,’’ an assembly code for FM8502 containing symbolic program names, labels, and abstract Piton objects. The principal output of this phase is a new program segment in which each program name is associated with its i-code. The formalization of this concept is embodied in the function **R->I** which is defined on page 202.
- Third, we replace all of the symbolic objects—in the stacks, in the assembled programs, and in the data segment—by bit vectors. This is called *link-assembling*. The i-code instructions are *assembled* into their machine code counterparts. Piton data objects are *linked* to (i.e., replaced by) the corresponding bit vectors. We are justified in calling this process ‘‘linking’’ because our compiled i-code uses Piton data objects (e.g., program addresses) for all references. Thus, the link phase is responsible for tying the system of programs and data together with absolute addresses. The formalization of this concept is embodied in the function **I->M** which is defined on page 181.

The organization of this chapter is as follows. We first present an example p-state and the corresponding m-state. This is offered primarily as proof that our implementation ultimately produces a ‘‘core image’’ as it is traditionally understood, despite the fact that the compiler and link-assembler are written in a computational logic. Next we sketch the implementation, primarily to establish terminology. Then we describe each of the three phases of the implementation: resource representation, compiling, and link-assembling.

6.1. An Example

Recall the program **BIG-ADD** and the initial state illustrated in Figure 3-2 on page 31.

The result of **LOAD**ing this state is FM8502 m-state shown in Figure 6-1.[10] There is absolutely no expectation on our part that this core dump is understandable. We only expect it to impress upon the reader that **LOAD** does indeed produce a core image, despite our early fascination with abstractions such as ‘‘resource representation.’’

---

[10]In this paper we use the standard notation for bit vectors, with the least significant digit to the right and the entire string of binary digits prefixed by the letter **B**. In Hunt’s formalization of FM8501—to which we actually map—bit vectors are shells constructed from **T** and **F** by the function **BITV** and the empty bit vector **(BTM)**. The least significant bit is the outermost in the nest. For example, the formal term denoted by **B011** is **(BITV T (BITV T (BITV F (BTM))))**. Readers familiar with our logic will recognize **’B011** as a **LITATOM**. To display the bit vectors in this report we simply defined the function in the logic that converts a **BITV** shell object into the appropriate literal atom.

Figure 6-1: The FM8502 Core Image for Big Number Addition

```
(M-STATE '(B00000000000000000000000000000010 B00000000000000000000000001110011
           B00000000000000000000000001110011 B00000000000000000000000001111110
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000000000000)
         F F F F
         '(B00000000000011111000001001000001 B00000000000011111000000000100010
           B00000000000011111000001001111000 B00000000000000000000000001100001
           B00000000000011111000001001111000 B00000000000000000000000001100101
           B00000000000011111000000010011000 B00000000000000000000000001101001
           B00000000000011111000001001101100 B00000000000011111000001001011000
           B00000000000000000000000000001101 B00000000000011111000000000001000
           B00000000000000000000000000010101 B00000000000011111000000010011000
           B00000000000000000000000001101010 B00000000000011111000000110011011
           B00000000000011111000000000001000 B00000000000000000000000000010010
           B00000000000011111000000001000001 B00000000000011111000000000111010
           B00000000000011111000000000011010 B00000000000011111000001001000001
           B00000000000011111000000000100010 B00000000000011111000001001011011
           B00000000000011111000001001011011 B00000000000011111000001001011011
           B00000000000011111000001001111000 B00000000000000000000000000000000
           B00000000000011111000000010011000 B00000000000000000000000000000000
           B00000000000000110000000010000010 B00000000000011111000001001101100
           B00000000000011111000000010011011 B00000000000011111000001001101100
           B00000000000011111000000010011000 B00000000000000000000000000000001
           B00000000000000110000000010000010 B00000000000011111000001001101100
           B00000000000011111000000010011011 B00000000000011111000001001101100
           B00000000000011111000000010011011 B00000000000011111000000010111011
           B00000000000010010010000101101011 B00000000000000100010000010000101
           B00000000000000110000000101111000 B00000000000000000000000000000001
           B00000000000011111000001001100100 B00000000000011111000000010011000
           B00000000000000000000000000000000 B00000000000000110000000010000010
           B00000000000011111000001001101100 B00000000000011111000000010011011
           B00000000000011111000000110011011 B00000000000011111000000010011000
           B00000000000000000000000000000010 B00000000000000110000000010000010
           B00000000000011111000001001101100 B00000000000000101000000101101011
           B00000000000011111000000010011000 B00000000000000000000000000000010
           B00000000000000110000000010000010 B00000000000011111000000110001011
           B00000000000011111000010010111011 B00000000000011111000000010011000
           B00000000000000000000000001011100 B00000000000000101100000000000100
           B00000000000011111000000010011000 B00000000000000000000000000000001
           B00000000000000110000000010000010 B00000000000011111000001001101100
           B00000000000011111000001001111000 B00000000000000000000000000000001
           B00000000000011111000000010011011 B00000000000000110000000101100100
           B00000000000011111000000010011000 B00000000000000000000000000000001
           B00000000000000110000000010000010 B00000000000011111000000110011011
           B00000000000011111000000010011000 B00000000000000000000000000000000
           B00000000000000110000000010000010 B00000000000011111000001001101100
           B00000000000011111000001001111000 B00000000000000000000000000000001
           B00000000000011111000000010011011 B00000000000000110000000101100100
           B00000000000011111000000010011000 B00000000000000000000000000000000
           B00000000000000110000000010000010 B00000000000011111000000110001011
           B00000000000011111000000000001000 B00000000000000000000000000100000
           B00000000000011111000000000001000 B00000000000000000000000001011110
           B00000000000011111000000001000001 B00000000000011111000000000111010
           B00000000000011111000000000011010 B00001110101101100111001101000010
           B10111001101111011111001001000001 B00000010100010101000010111001111
           B00000000000000000000000000000000 B11010101010110001100000100001000
           B11101101001101010101101000101000 B00000000011101010110001111000010
           B00000000000000000000000000000000 B00000000000000000000000000000100
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000001110011
           B00000000000000000000000000000010 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000000000000
           B00000000000000000000000000000000 B00000000000000000000000001101011
           B00000000000000000000000001110110 B00000000000000000000000001111110))
```

6.2. A Sketch of the FM8502 Implementation

Our implementation of Piton partitions the memory of FM8502 into three parts. The first holds the binary programs and is called the *program segment*. The second holds the data segment of the Piton machine and is called the *user data segment* of the FM8502 memory. The third holds the control and temporary stacks and three words of data used to implement the stack resource instructions. This segment of the FM8502 memory is called the *system data segment*. In addition, the implementation uses six of the eight registers of FM8502 and the four condition code flags.

6.2.1. The Registers

We give symbolic names to six of the FM8502's registers. The actual register numbers allocated are 0-5, in the order listed.

- *pc* (program counter).
- *cfp* (control stack frame pointer): determines the extent of the topmost frame on the control stack.
- *csp* (control stack pointer): addresses the top of the control stack.
- *tsp* (temporary stack pointer): addresses the top of the temporary stack.
- *x* and *y*: used as *temporaries* in the machine code implementing Piton instructions: no assumptions are made about the values of these registers when (the machine code for) a Piton instruction begins execution and no promises are offered about the values at the conclusion of execution.

6.2.2. The Condition Codes

Recall the four condition code registers of FM8502: carry, overflow, negative, and zero. These can be set by any instruction according to the operation performed by the alu during that instruction. The condition codes are tested by the conditional move instructions.

The implementation of certain Piton instructions, namely the ‘‘test and jump’’ family and the ‘‘arithmetic with carry’’ family, load and test the condition code registers of FM8502. However, these registers are also treated as temporaries in the machine code.

6.2.3. The Program Segment

The program segment of the FM8502 memory is logically divided into separate areas. Each area corresponds to a Piton program. The contents of such an area is the binary machine code obtained by compiling the corresponding Piton program and then link-assembling it. To compile a Piton program we compile each individual instruction in it, concatenate the results, and sandwich the resulting code between a ‘‘prelude’’ and a ‘‘postlude’’ that implements procedure call and return. We are much more specific later. However, the generated code is easily imagined once we have explained how the abstract stacks and data areas of Piton are concretely implemented.

6.2.4. User Data Segment

The user data segment of the FM8502 machine is isomorphic to the data segment of the Piton machine. The data segment is allocated in the FM8502 memory space immediately above the program segment. Each array in the data segment is allocated a block of words as long as the array. The arrays are laid out successively, each immediately adjacent to its neighbors.

6.2.5. The System Data Segment

Immediately above the data segment we allocate the system data segment. This segment contains five ‘‘areas,’’ comparable to the areas (arrays) of the user data segment. In the *control stack area* we represent the Piton control stack, using, in addition, the **`cfp`** and **`csp`** registers. In the *temporary stack area* we represent the Piton temporary stack, using, in addition, the **`tsp`** register. The other three system data areas contain resource bounds used in the implementation of such instructions as **`JUMP-IF-TEMP-STK-EMPTY`**.

The amount of space allocated to each stack is one greater than the maximum stack size for that stack, as specified in the Piton state.

Because of the addressing modes in FM8502 it was decided that the stacks should grow ‘‘downwards’’ (i.e., toward absolute address 0). When a push is performed, the stack pointer is decremented and then the operand is deposited into the memory location indicated by the new stack pointer; when a pop is performed, the contents of the indicated memory location is fetched and then the pointer is incremented. When a stack is empty its stack pointer addresses the high word in its area; the contents of this word is never read or written and hence the allocation of the word is actually a waste of one word.[11]

6.2.5.1. The Temporary Stack

Consider the following Piton temporary stack containing four Piton data objects:

(obj_1
obj_2
obj_3
obj_4**).**

obj_1 is the topmost element of the stack. Suppose the maximum temporary stack size is 6.

At the FM8502 level, this temporary stack is represented by a block of 7 words in the system data segment, together with the **`tsp`** register. Suppose, for concreteness, that the temporary stack data area is allocated beginning at absolute address 1000. In Figure 6-2 we show how the above stack is represented. Of course, the contents of the memory locations are not the Piton objects themselves but bit vectors representing them. We deal with this problem later. A push on the stack above would decrement **`tsp`** and write into address **`1001`**. A pop would fetch from the current **`tsp`** and then increment **`tsp`** to **`1003`**. If the stack is popped four times it is empty and **`tsp`** would be **`1006`**. Because we allocated an extra word to the temporary stack area, the empty stack pointer is still an address into the temporary stack area. Because we will never pop the empty stack, the contents of the extra word is irrelevant.

---

[11]The implementation could have made the empty stack pointer be an address ‘‘just outside’’ the stack area. However, by allocating the extra word we slightly simplified the correctness proof because we maintained the invariant that the stack pointers were always addresses into the appropriate system data area.

Figure 6-2: An FM8502 Temporary Stack

memory address	contents	**registers** name	contents
1006	*unused*	**tsp**	**1002**
1005	obj_4		
1004	obj_3		
1003	obj_2		
1002	obj_1		
1001	*inactive*		
1000	*inactive*		

6.2.5.2. The Control Stack

The Piton control stack is a stack of frames, each frame containing the bindings for the local variables of the current subroutine and the return program counter. At the FM8502 level, each frame contains the values of the local variables (but not the names) and the return program counter; the machine code references the values by position within the frame. Because the FM8502 frames are just blocks of words laid out consecutively in the control stack area, each frame contains an additional word that points to the ‘‘beginning’’ of the previous frame.

We explain the exact layout of the control stack by example. Below we show a Piton control stack containing three frames.

```
(LIST
  (LIST '((X . valx)          ;Frame 1 - current
          (Y . valy)
          (Z . valz))
        ret-pc1)
  (LIST '((A . vala)          ;Frame 2
          (B . valb))
        ret-pc2)
  (LIST '((U . valu)          ;Frame 3
          (V . valv))
        ret-pc3))
```

Suppose the maximum control stack size is 16. Then the stack above is represented as a block of 17 consecutive words in the system data segment, together with two registers, the **csp** register and the **cfp** register.

For the sake of concreteness, suppose the control stack area begins at address 5000. Then the relevant portion of the FM8502 state is shown in Figure 6-3.

Frame 1 of the Piton stack is represented in memory locations **5003-5007** of Figure 6-3. In particular, the return program counter, $ret\text{-}pc_1$, is in ‘‘first’’ word of the top frame (where we enumerate the words in the same order in which they were pushed), location **5007**. The second word of the top frame, location **5006**, contains the address of the frame that was current at the time this frame was built. We call this the *old cfp* word of the frame, for reasons that will become apparent. The bindings are in the last n words of the frame, locations **5003-5005**.

Figure 6-3: An FM8502 Control Stack

memory		**registers**	
address	contents	name	contents
5016	*unused*	**cfp**	**5006**
5015	$ret\text{-}pc_3$	**csp**	**5003**
5014	**5014**		
5013	val_v		
5012	val_u		
5011	$ret\text{-}pc_2$		
5010	**5014**		
5009	val_b		
5008	val_a		
5007	$ret\text{-}pc_1$		
5006	**5010**		
5005	val_z		
5004	val_y		
5003	val_x		
5002	*inactive*		
5001	*inactive*		
5000	*inactive*		

The **csp** register contains **5003**, the ‘‘top’’ of the control stack. The values of the successive local variables are obtained by indexing up from **csp**. The value of the 0^{th} local is at **csp**+0, i.e., **5003**, the value of the 1^{st} local is at **csp**+1, i.e., **5004**, etc.

The **cfp** register contains **5006**. This is the address of the old cfp word in the top frame. That word contains the address of the old cfp word in the previous frame. By starting at **cfp** and threading through the old cfp words one can identify each frame.

Frame 2 of the Piton stack, the one extant when frame 1 was created, is in locations **5008-5011**. Frame 3 is in locations **5012-5015**.

Locations **5000-5002** of the area above are currently unused but will be filled if a new frame is pushed. Location **5016**, the high word in the control stack area, is wasted.

Finally, it should be noted that the deepest frame on the stack, frame 3 in our example, is somewhat pathological. First, its return program counter is irrelevant. This is a peculiarity of Piton, not our implementation. When **RET** is executed in the context of this frame (the top-level entry into the Piton system) the Piton machine halts rather than transfer control out of Piton. So the user of Piton, who starts the machine on some initial state, may select any value he wishes for the initial return pc. Second, the old cfp word of that frame is irrelevant for the same reason. In our implementation, the old cfp word of the last frame always points to itself.

6.2.5.3. Stack Resource Limits

In addition to the two stack areas, the system data segment contains three words used to implement the stack resource instructions. These three words each comprise a named area within the system data segment.

The names and contents of the three additional areas are

- *full control stack address*: the address of the lowest word in the control stack area. In our example, it is **`5000`**. When the control stack pointer is equal to this value, the control stack is full; an additional push would cause the pointer to overflow the control stack area.
- *full temporary stack address*: the address of the lowest word in the temporary stack area. In our example, it is **`1000`**.
- *empty temporary stack address*: the address of the highest word in the temporary stack area. In our example, it is **`1006`**. When the temporary stack pointer is equal to this value, the temporary stack is empty.

This completes our sketch of the implementation. In the next three sections we discuss each phase of the implementation in more detail.

6.3. Resource Representation

The first phase of **`LOAD`**, **`P->R`**, constructs a symbolic description of the layout of the system data segment and the initial values of the registers. We describe these resources symbolically first, instead of mapping directly to bit vectors, because the stacks and registers contain Piton data objects (such as data and program addresses) whose bit vector representations are not known until link time.

6.3.1. The System Data Segment and System Addresses

The symbolic description of the system data segment is a list structure similar to Piton's data segment. In particular, the system data segment is described by a list of length five. Each element describes a *system data area* by listing the *name* of the area and the *array* of objects associated with it. We give the following literal atom names to the areas in the system data segment: **`CSTK`**, **`TSTK`**, **`FULL-CTRL-STK-ADDR`**, **`FULL-TEMP-STK-ADDR`**, and **`EMPTY-TEMP-STK-ADDR`**.

Intuitively, the array associated with each area is just a list of tagged Piton objects. This is not completely accurate because it is necessary for the control stack to contain addresses into itself, namely the old cfp words, and no Piton data object addresses the system data segment. We therefore introduce an eighth type of object, called a *system data address*, which is a pair of the form **`(name . n)`**, where **`name`** is one of the area names above and **`n`** is a natural number less than the length of the associated area.[12] Thus, **`(FULL-CTRL-STK-ADDR . 0)`** is the symbolic address of the full control stack address word in the system data segment. Recall that that word contains the address of the lowest word in the control stack area. In fact we can now write that address down symbolically too. It is **`(CSTK . 0)`**.

System addresses will be tagged with the literal atom **`SYS-ADDR`**, exactly analogous to the way data addresses are tagged with **`ADDR`**. Thus, if we write **`(SYS-ADDR (CSTK . 25))`** we are referring to the 25th word in the control stack area in the system data segment. If we write **`(ADDR (CSTK . 25))`** we are referring to the 25th word in a global data area named **`CSTK`** (supposing one exists) in the Piton data segment. We usually omit the tag in informal text and simply say ''the system address **`(CSTK . 25)`**'' or ''the data address **`(CSTK . 25)`**.''

---

[12]The implied motivation for system addresses was to represent the old cfp words in the control stack. We have implemented a general purpose system addressing notation because system addresses are used throughout the implementation. For example, they may be found in the compiler output.

The arrays associated with each of the five area names are simply lists of either Piton objects or system addresses. The previous section sketching the layout of the stacks should make it clear how we load these arrays.

We conclude with a simple example. Suppose the maximum control stack size is 11 and the maximum temporary stack size is 8. Suppose the control stack of the Piton state is

```
((((X . (NAT 0))              ; Frame 1 - current
   (Y . (NAT 1))
   (Z . (NAT 2)))
  (PC (MAIN . 4)))
 (((A . (INT 1))              ; Frame 2
   (B . (INT 2)))
  (PC (MAIN . 0)))).
```

Suppose the temporary stack is

```
((ADDR (A . 3))               ; topmost element
 (BOOL T)
 (NAT 27)).
```

The symbolic description of the corresponding system data segment is shown in Figure 6-4.

Figure 6-4: A System Data Segment

```
                                   offset     comment
((CSTK (NAT 0)                     ; 0        inactive
       (NAT 0)                     ; 1        value of X - Frame 1
       (NAT 1)                     ; 2        value of Y
       (NAT 2)                     ; 3        value of Z
       (SYS-ADDR (CSTK . 8))       ; 4        old cfp
       (PC (MAIN . 4))             ; 5        return pc
       (INT 1)                     ; 6        value of A - Frame 2
       (INT 2)                     ; 7        value of B
       (SYS-ADDR (CSTK . 8))       ; 8        old cfp
       (PC (MAIN . 0))             ; 9        return pc
       (NAT 0))                    ;10        unused
 (TSTK (NAT 0)                     ; 0        inactive
       (NAT 0)                     ; 1        inactive
       (NAT 0)                     ; 2        inactive
       (NAT 0)                     ; 3        inactive
       (NAT 0)                     ; 4        inactive
       (ADDR (A . 3))              ; 5        topmost element
       (BOOL T)                    ; 6
       (NAT 27)                    ; 7        btm-most element
       (NAT 0))                    ; 8        unused
 (FULL-CTRL-STK-ADDR (SYS-ADDR (CSTK . 0)))
 (FULL-TEMP-STK-ADDR (SYS-ADDR (TSTK . 0)))
 (EMPTY-TEMP-STK-ADDR (SYS-ADDR (TSTK . 8))))
```

6.3.2. The Registers

The resource representation phase also specifies the symbolic value of each of the registers. This is necessary since the registers participate in the representation of the stacks. The **pc** register is exactly the same as in the Piton state. The **cfp**, **csp** and **tsp** registers are set to the appropriate system addresses. For the system data segment in Figure 6-4 the register values are

```
cfp:      (SYS-ADDR (CSTK . 4))
csp:      (SYS-ADDR (CSTK . 1))
tsp:      (SYS-ADDR (TSTK . 5))
```

The initial values of the **x** and **y** registers and of the condition code flags are irrelevant.

The system addresses, along with all the other data objects in the resource representation description, will be turned into bit vectors by the link phase, when we know where, in absolute terms, the system data segment will be located.

6.4. Compiling

The next phase of the **LOAD** process, **R->I**, is compilation. The compiler scans the program segment of the Piton state and pairs each program name with the assembly code for that program. Observe that compilation of each program is independent of the other programs and of the other components of the Piton state. It is the link phase that worries about references between programs and data.

The assembly code for a Piton program is logically divided into three parts. The first part, called the *prelude*, is executed as part of subroutine **CALL**. The prelude builds the new control stack frame for the invocation and removes the actuals from the temporary stack. The second part, called the *body*, is the translation of the successive instructions in the body of the Piton program. The third part, called the *postlude*, is executed as part of the **RET** instruction and pops the top frame off the control stack, returning control to the indicated pc.

We exhibit a simple compilation below. Consider the following silly Piton program:

```
(DEMO (X Y Z)                      ; Formals X, Y, and Z
      ((A (INT -1))                ; Temporaries A and I
       (I (NAT 2)))
      (PUSH-LOCAL Y)               ; pc 0
      (PUSH-CONSTANT (NAT 4))      ; pc 1
      (ADD-NAT)                    ; pc 2
      (RET))                       ; pc 3
```

The program, named **DEMO**, has three formals and two locals. It adds 4 to the value of the second formal and leaves the result on the stack.

The output of the compiler on this program is shown in Figure 6-5. The output is in a symbolic, annotated version of FM8502 machine code, called ‘‘i-code.’’ I-code is similar to assembly code in that instructions are written in symbolic form and translate 1:1 into FM8502 machine code. We discuss i-code at greater length later.

In Figure 6-5 we have printed the compiler’s output in a four column format to make its structure clearer.

The first column consists entirely of labels. Note that our symbolic code uses the same **DL** construct used by Piton. The second column contains i-code instructions and data.[13] The third column contains the Piton source code used as input to the compiler. The compiler uses the comment field of the **DL** construct to annotate each block of i-code with the high level instruction that generated it. The fourth column consists of manually inserted comments that enumerate the successive values of the i-code program counter.

Figure 6-5: Compiler Output for DEMO

```
                     i-code               Piton
      label          instruction          instruction                pc
---------------------------------------------------------------------------
(DEMO
 (DL (DEMO PRELUDE)                      (PRELUDE)
                     (CPUSH_CFP))                                  ; 0
                     (MOVE_CFP_CSP)                                ; 1
                     (CPUSH_*)                                     ; 2
                     (NAT 2)                                       ; 3
                     (CPUSH_*)                                     ; 4
                     (INT -1)                                      ; 5
                     (CPUSH_<TSP>+)                                ; 6
                     (CPUSH_<TSP>+)                                ; 7
                     (CPUSH_<TSP>+)                                ; 8
 (DL (DEMO . 0)                          (PUSH-LOCAL Y)
                     (MOVE_X_*))                                   ; 9
                     (NAT 1)                                       ;10
                     (ADD_X{N}_CSP)                                ;11
                     (TPUSH_<X{S}>)                                ;12
 (DL (DEMO . 1)                          (PUSH-CONSTANT (NAT 4))
                     (TPUSH_*))                                    ;13
                     (NAT 4)                                       ;14
 (DL (DEMO . 2)                          (ADD-NAT)
                     (TPOP_X))                                     ;15
                     (ADD_<TSP>{N}_X{N})                           ;16
 (DL (DEMO . 3)                          (RET)
                     (JUMP_*))                                     ;17
                     (PC (DEMO . 4))                               ;18
 (DL (DEMO . 4)                          (POSTLUDE)
                     (MOVE_CSP_CFP))                               ;19
                     (CPOP_CFP)                                    ;20
                     (CPOP_PC))                                    ;21
```

The prelude for the code in Figure 6-5 consists of instructions **0** through **8**. The prelude is labelled by the label **(DEMO PRELUDE)**, a list object in the logic and guaranteed to be unique among the labels used by the assembler. All labels, of course, are eventually removed by the linker and merely serve as unique entries in the link table.

The prelude builds a new frame on the control stack. The construction of the frame begins before the prelude is entered, when the **CALL** instruction is executed. **CALL** pushes the return program counter onto

[13]FM8502 machine code instructions can take ‘‘immediate’’ data from the next word in the instruction stream using the post-increment addressing mode with the program counter register, as explained later. In our i-code, such double-word instructions have names ending in a ‘‘*****.’’

the control stack and jumps to the prelude of the called subroutine. Recall that the first word in the new frame is the return program counter (which has just been pushed). The prelude builds the rest of the frame by pushing more words onto the control stack.

Below we display the prelude for **DEMO**.

```
(CPUSH_CFP)                                        ; 0
(MOVE_CFP_CSP)                                     ; 1
(CPUSH_*)                                          ; 2
(NAT 2)                                            ; 3
(CPUSH_*)                                          ; 4
(INT -1)                                           ; 5
(CPUSH_<TSP>+)                                     ; 6
(CPUSH_<TSP>+)                                     ; 7
(CPUSH_<TSP>+)                                     ; 8
```

At instruction **0** the current frame pointer register, **cfp**, is pushed onto the control stack. That sets the old cfp word of the frame. At instruction **1** the current **csp**, which now points to the old cfp word just pushed, is moved into the **cfp** register. Thus, the **cfp** register points to the old cfp word of the frame under construction. The instruction at **2** is a double-word instruction that pushes the natural number **2** onto the control stack. This is the initial value of the temporary variable **I**. At instruction **4** the initial value of the temporary variable **A** is pushed. The last three instructions of the prelude, **6**-**8**, each pop one thing off the temporary stack and push the result onto the control stack. These instructions move the actuals into the new frame. This completes the construction of the frame. Note that the **csp** register points to the top of the stack and the **cfp** register points to the old cfp word as required.

The first executable Piton instruction in the program, **(PUSH-LOCAL Y)**, is compiled as instructions **9**-**12**. That block of code is labelled in the assembly language by **(DEMO . 0)** which happens to be the Piton program address of the Piton instruction for which the code was generated. *Each program address at the Piton level is defined as a label in the i-code.* It is via this identification of Piton program address objects with i-code labels that the linker is able to replace each **PC** object by its absolute address. However, there may be objects tagged **PC** in our i-code that are not legal program address objects in Piton, e.g., **(DEMO PRELUDE)**. In general, the tag **PC** in i-code means ‘‘i-code label,’’ not Piton program counter.

The instructions for **(PUSH-LOCAL Y)** are

```
(MOVE_X_*)                                         ; 9
(NAT 1)                                            ;10
(ADD_X{N}_CSP)                                     ;11
(TPUSH_<X{S}>)                                     ;12
```

The basic idea is to fetch a certain element from the top frame and push it onto the temporary stack. The element is the one in the slot for the local variable **Y**, which is in position **1** of the locals. Recall that the locals are numbered from **0** and **X** is thus the **0**th local of the program; the value of **X** is thus at the address indicated by **csp**. The value of **Y** is at the address one greater than **csp**. The code above may be explained as follows: (**9**) move the index, **1**, of the local variable into the **x** register. (**11**) add the contents of the **csp** register to **x** and store the result in **x**; this is the address of the appropriate slot in the control stack. (**12**) fetch indirect through the **x** register and push the result onto the temporary stack.

In our symbolic code, we use angle brackets, e.g., **<X>**, around a register to indicate register-indirect addressing mode. We use set braces, e.g., **{S}**, to indicate the type of object in the register. Thus, the instruction **ADD_X{N}_CSP** means ‘‘add the contents of **csp** to *the natural number in* **x**’’ and

TPUSH_<X{S}> means ‘‘indirect through *the system data address in* **X**.’’ We discuss the data type annotations later.

The next instruction in the Piton program is **(PUSH-CONSTANT (NAT 4))** and the code generated is

```
(TPUSH_*)                                              ;13
(NAT 4)                                                ;14
```

This code pushes the natural number 4 onto the temporary stack.

The next Piton instruction is **(ADD-NAT)**, which is supposed to pop two naturals off the temporary stack, add them together, and push the result. The generated code is

```
(TPOP_X)                                               ;15
(ADD_<TSP>{N}_X{N})                                    ;16
```

The code pops one thing off the temporary stack into **X**. Then it adds the natural in **X** to the natural fetched indirect through **tsp** (the top item on the stack), and deposits the result indirect through **tsp** (back onto the top of the stack).

The last executable instruction in the Piton code is the return instruction. It compiles into

```
(JUMP_*)                                               ;17
(PC (DEMO . 4))                                        ;18
```

This is just an unconditional jump to the label **(DEMO . 4)**, which is where the postlude is located.[14]

The postlude is

```
(MOVE_CSP_CFP)                                         ;19
(CPOP_CFP)                                             ;20
(CPOP_PC))                                             ;21
```

The postlude must remove the top frame from the stack, restore the **cfp** register to the value it had at the time of the **CALL**, and restore the program counter to the return pc. The first instruction moves the contents of the **cfp** register into **csp**. This effectively pops all the bindings of this frame and makes the top of the stack be the old cfp word. The next instruction pops the control stack into the **cfp** register, restoring **cfp** and exposing the return pc at the top of the stack. The last instruction pops the control stack into the **pc**, completing the return and removing the last vestige of the now popped frame.

Now consider the following segment of a Piton program **MAIN** that calls **DEMO** on the address of the 25th element of the array **DELTA1**, the natural number 17, and the Boolean value **T**. Suppose the **CALL** instruction is located at program address **(MAIN . 3)**.

```
(PUSH-CONSTANT (ADDR (DELTA1 . 25)))
(PUSH-CONSTANT (NAT 17))
(PUSH-CONSTANT (BOOL T))
(CALL DEMO)
```

The i-code generated for this segment is shown below. We have stripped out the label definitions.

---

[14]Our compiler has much room for improvement. The instruction at **17** jumps to the next executable instruction, and so could be eliminated. We do not do any such optimizations.

```
(TPUSH_*)                ;Push first actual on temp
(ADDR (DELTA1 . 25))
(TPUSH_*)                ;Push second actual on temp
(NAT 17)
(TPUSH_*)                ;Push third actual on temp
(BOOL T)
(CPUSH_*)                ;Push return pc on ctrl
(PC (MAIN . 4))
(JUMP_*)                 ;Jump to (DEMO PRELUDE)
(PC (DEMO PRELUDE))
```

Observe that all addresses, both program and data, are represented in the i-code by *extended data objects*—i.e., either Piton data objects, system data address objects, or i-code labels tagged **PC**. This is true regardless of how the address originated. For example, **(DELTA . 25)** was originally a data object in the source program. The reference to **DEMO** occurred in the **CALL** instruction and has been transformed into the i-code data object **(DEMO PRELUDE)** of type **PC**. The return pc **(MAIN . 4)** above was only implicit in the source program.

The next example compilation illustrates our handling of labels. Consider the following program, **PTZ** (‘‘Pop till Zero’’), which pops the temporary stack until it pops a **0**.

```
(PTZ NIL
     NIL
 (DL LOOP ()
     (TEST-NAT-AND-JUMP ZERO END))
     (JUMP LOOP)
 (DL END ()
     (RET)))
```

In Figure 6-6 we show the compiler output for **PTZ**.

Let us look carefully at the code generated for the **TEST-NAT-AND-JUMP** instruction. It is supposed to pop the stack and jump to the label **END** if the result is the natural number 0. The i-code is in locations **2-5**. The first instruction pops the stack into the **y** register and sets the **z** condition code (according to whether the result is 0). Next, we move into the **x** register the **PC** data object **(PTZ . 2)**. Then we jump to the contents of **x** if the **z** condition code is set. Inspection will show that **(PTZ . 2)** is the i-code label marking the point labelled **END** in the Piton source code.

Similarly, observe the compilation of the **(JUMP LOOP)** instruction. The i-code is at locations **6-7** above. It reads: Jump to i-code label **(PTZ . 0)**.

In general, references to labels in Piton are compiled into references to **PC** type data objects.

As noted above, all program and data addresses, no matter how they originate, are explicitly mentioned extended data objects (of type **PC**, **ADDR**, **SYS-ADDR**, or **SUBR**) in the i-code produced by the Piton compiler. It is the job of the linker, discussed next, to replace these objects by the corresponding absolute addresses. Until this is done, the code and the data segment are relocatable.

Recall that we have introduced a new type of object, the system address, which exists in our implementation of Piton but is not one of the types in Piton. When the compiler must make a reference to a word in the system data segment it uses these symbolic addresses so that the code is relocatable with respect to where the system data segment is laid out.

Figure 6-6: Compiler Output for PTZ

```
label                     i-code                  Piton
                          instruction             instruction             pc
--------------------------------------------------------------------------------
(PTZ
 (DL (PTZ PRELUDE)                                     (PRELUDE)
                     (CPUSH_CFP))                                         ; 0
                     (MOVE_CFP_CSP)                                       ; 1
 (DL (PTZ . 0)                                     (DL LOOP NIL
                                                       (TEST-NAT-AND-JUMP ZERO
                                                                     END))
                     (TPOP{N}_<Z>_Y))                                     ; 2
                     (MOVE_X_*)                                           ; 3
                     (PC (PTZ . 2))                                       ; 4
                     (JUMP-Z_X)                                           ; 5
 (DL (PTZ . 1)                                         (JUMP LOOP)
                     (JUMP_*))                                            ; 6
                     (PC (PTZ . 0))                                       ; 7
 (DL (PTZ . 2)                                     (DL END NIL (RET))
                     (JUMP_*))                                            ; 8
                     (PC (PTZ . 3))                                       ; 9
 (DL (PTZ . 3)                                         (POSTLUDE)
                     (MOVE_CSP_CFP))                                      ;10
                     (CPOP_CFP)                                           ;11
                     (CPOP_PC))                                           ;12
```

To illustrate the use of these internal addresses, consider the compilation of

```
(JUMP-IF-TEMP-STK-FULL ERROR)
```

and suppose that the label **ERROR** is defined at **PC (MAIN . 152)**. The i-code generated for this instruction is

```
(MOVE_X_TSP)                                            ;0
(MOVE_Y_*)                                              ;1
(SYS-ADDR (FULL-TEMP-STK-ADDR . 0))                     ;2
(MOVE_Y_<Y{S}>)                                         ;3
(SUB_<Z>_X{S}_Y{S})                                     ;4
(MOVE_X_*)                                              ;5
(PC (MAIN . 152))                                       ;6
(JUMP-Z_X)                                              ;7
```

The code first puts the temporary stack pointer into the **x** register so we can do some arithmetic on it. Then, in lines **1-3** above, the code fetches into the **y** register the address of the first word in the temporary stack area. This is done in two instructions. First (in lines **1-2**), we load into **y** the system address **(FULL-TEMP-STK-ADDR . 0)**. In our implementation, the contents of this address is the address of the first word in the temporary stack area. On line **3** we fetch indirect through the system address in **y** and put the result in **y**. This loads **y** with the address of the first location on the temporary stack area.[15]

---

[15] In fact, the code for **JUMP-IF-TEMP-STK-FULL** could be shortened by eliminating the indirection through **FULL-TEMP-STK-ADDR** and simply loading **(SYS-ADDR (TSTK . 0))** into the **y** register. We introduced **FULL-TEMP-STK-ADDR** primarily to force our proof to deal with such common implementation invariants as ‘‘the contents of this fixed address is the address of this more fluid boundary.’’

On line **4** we subtract the system address in **y** from the system address in **x** and set the **z** condition code register. Thus, the **z** register is **T** iff **x** and **y** were equal. On lines **5-6** we move into **x** the address to which we wish to jump. On line **7** we jump if **z** is true.

The examples shown in this section are fairly representative of the code generated by the Piton compiler. A complete listing of the compiler—which essentially is just an enumeration of the Piton instructions and the code generators for them—is given in the appendix.

6.5. The Link-Assembler

Traditionally, compilers produce relocatable assembly code, which is then turned into relocatable machine code by an ‘‘assembler’’, and then into absolute machine code by a ‘‘linker.’’ In addition, assemblers and linkers must traditionally consider the user’s data declarations and initialization too. We do not follow this paradigm rigidly but the basic concepts are still present in our ‘‘link-assembler.’’

By the time the link-assembler is invoked the first two phases of **LOAD** have been carried out: the resource representation phase has produced symbolic descriptions of the registers and the system data segment; the compiler has produced the i-code version of the program segment. The user data segment is symbolically described by the Piton data segment. The job of our ‘‘link-assembler,’’ **I->M**, is to replace the symbolic instructions and data objects by concrete bit vectors.

To do so, the link-assembler first builds a collection of ‘‘link tables’’ which indicate where each program, label, system data area and user data area are located in absolute terms. This is done by the function **I-LINK-TABLES** which is defined on page 181. Note that by the time we build the link tables, the three segments of the FM8502 memory are symbolically described by the i-code program segment, the Piton data segment, and the system data segment. All three of these symbolic descriptions have the same form: each is a list of pairs consisting of the name of the program or area and an array listing the contents of the area. Each element of the array is either an (optionally labelled) i-code instruction or an extended data object. Each element can be mapped to a single word in FM8502. Thus, the number of words to be allocated to each area is just the length of the associated array. The absolute location of each name can be determined by summing the lengths of the areas preceding the definition of the name. The absolute location of each label can be similarly determined by counting the number of items in each i-code program preceding the label definition.

Once the link tables have been created, the link-assembler scans each of the three memory segments in turn. Each i-code instruction is replaced by the corresponding FM8502 machine code instruction, by a function which can be thought of as the basic component of an assembler. Each data object is replaced by the corresponding bit vector, using the link tables as appropriate.

We present the link-assembler in three subsections. First, we describe how individual i-code instructions are assembled into FM8502 machine code instructions. Then we discuss how we generate the link tables. Finally, we describe how we transform each of the data objects.

6.5.1. The Instruction Assembler

The instruction assembler converts a single i-code instruction into an FM8502 machine code instruction, i.e., a 32-bit wide bit vector. The conversion is done in two steps and explicitly involves an assembly language for FM8502. In essence, each i-code instruction is taken as a ‘‘pseudo-instruction’’ that is mapped first into an assembly instruction and then into a bit vector. The formalization of this concept is embodied in the function **LINK-INSTR-WORD** which is defined on page 195.

6.5.1.1. Expanding I-code into Assembly Code

Each i-code instruction is of the form **(opcode)**, where **opcode** is a literal atom in the logic and completely describes the instruction. Our i-code instructions do not have operands, even though their names e.g., **ADD_<TSP>{N}_X{N}**, suggest more structure. We explain the design of i-code when we discuss the proof of the correctness of our implementation.

The following table, which is an association list, is the map from i-code opcodes to assembly instructions. This table lists all i-code opcodes and gives a good idea of the structure of our assembly code for FM8502.

Map from I-code to Assembly Code

```
        i-code                              assembly code
-------------------------------------------------------------------------
((ADD_<C>_X_X{N}               (ADD (C) X X))
 (ADD_<TSP>_<TSP>{V}           (ADD () (TSP) (TSP)))
 (ADD_<TSP>_<TSP>{N}           (ADD () (TSP) (TSP)))
 (ADD_<TSP>{A}_X{N}            (ADD () (TSP) X))
 (ADD_TSP_*{N}                 (ADD () TSP (PC +1)))
 (ADD_TSP_X{N}                 (ADD () TSP X))
 (ADD_<TSP>{I}_X{I}            (ADD () (TSP) X))
 (ADD_<TSP>{N}_X{N}            (ADD () (TSP) X))
 (ADD_PC_X{N}                  (ADD () PC X))
 (ADD_X_X{N}                   (ADD () X X))
 (ADD_X{N}_CSP                 (ADD () X CSP))
 (ADDC_<C>_X{N}_Y{N}           (ADDC (C) X Y))
 (ADDC_<V>_X{I}_Y{I}           (ADDC (V) X Y))
 (AND_<TSP>{V}_X{V}            (AND () (TSP) X))
 (AND_<TSP>{B}_X{B}            (AND () (TSP) X))
 (ASR_<C>_<TSP>_<TSP>{B}       (ASR (C) (TSP) (TSP)))
 (CPOP_CFP                     (MOVE () CFP (CSP +1)))
 (CPOP_PC                      (MOVE () PC (CSP +1)))
 (CPUSH_*                      (MOVE () (-1 CSP) (PC +1)))
 (CPUSH_<TSP>+                 (MOVE () (-1 CSP) (TSP +1)))
 (CPUSH_CFP                    (MOVE () (-1 CSP) CFP))
 (DECR_<TSP>_<TSP>{I}          (DECR () (TSP) (TSP)))
 (DECR_<TSP>_<TSP>{N}          (DECR () (TSP) (TSP)))
 (INCR_<TSP>_<TSP>{I}          (INCR () (TSP) (TSP)))
 (INCR_<TSP>_<TSP>{N}          (INCR () (TSP) (TSP)))
 (INCR_Y_Y{N}                  (INCR () Y Y))
 (INT-TO-NAT                   (MOVE () X X))
 (JUMP-N_X                     (MOVE-N () PC X))
 (JUMP-NN_X                    (MOVE-NN () PC X))
 (JUMP-NZ_X                    (MOVE-NZ () PC X))
 (JUMP-Z_X                     (MOVE-Z () PC X))
 (JUMP_*                       (MOVE () PC (PC)))
 (JUMP_X{SUBR}                 (MOVE () PC X))
 (LSR_<C>_X_X{N}               (LSR (C)  X X))
 (LSR_<TSP>_<TSP>{V}           (LSR () (TSP) (TSP)))
 (MOVE-C_<TSP>_*               (MOVE-C () (TSP) (PC +1)))
 (MOVE-V_<TSP>_*               (MOVE-V () (TSP) (PC +1)))
```

```
(MOVE-Z_<TSP>_*             (MOVE-Z () (TSP) (PC +1)))
(MOVE-N_X_*                 (MOVE-N () X (PC +1)))
(MOVE_<TSP>_*               (MOVE () (TSP) (PC +1)))
(MOVE_<X{A}>_<TSP>          (MOVE () (X) (TSP)))
(MOVE_<X{S}>_<TSP>          (MOVE () (X) (TSP)))
(MOVE_CFP_CSP               (MOVE () CFP CSP))
(MOVE_CSP_CFP               (MOVE () CSP CFP))
(MOVE_X_*                   (MOVE () X (PC +1)))
(MOVE_X_<X{S}>              (MOVE () X (X)))
(MOVE_X_TSP                 (MOVE () X TSP))
(MOVE_X_X                   (MOVE () X X))
(MOVE_Y_*                   (MOVE () Y (PC +1)))
(MOVE_Y_<Y{S}>              (MOVE () Y (Y)))
(MOVE_Y_TSP                 (MOVE () Y TSP))
(NEG_<TSP>_<TSP>{I}         (NEG () (TSP) (TSP)))
(NOT_<TSP>_<TSP>{V}         (NOT () (TSP) (TSP)))
(OR_<TSP>{V}_X{V}           (OR () (TSP) X))
(OR_<TSP>{B}_X{B}           (OR () (TSP) X))
(SUB_<C>_<TSP>{A}_X{A}      (SUB (C) (TSP) X))
(SUB_<C>_<TSP>{N}_X{N}      (SUB (C) (TSP) X))
(SUB_<NV>_<TSP>{I}_X{I}     (SUB (N V) (TSP) X))
(SUB_<TSP>{A}_X{N}          (SUB () (TSP) X))
(SUB_X{S}_Y{N}              (SUB () X Y))
(SUB_<TSP>{I}_X{I}          (SUB () (TSP) X))
(SUB_<TSP>{N}_X{N}          (SUB () (TSP) X))
(SUB_<TSP>{S}_X{S}          (SUB () (TSP) X))
(SUB_<Z>_X{S}_Y{S}          (SUB (Z) X Y))
(SUBB_<C>_X{N}_Y{N}         (SUBB (C) X Y))
(SUBB_<V>_X{I}_Y{I}         (SUBB (V) X Y))
(TPOP_<C>_X                 (MOVE (C) X (TSP +1)))
(TPOP_<X{A}>                (MOVE () (X) (TSP +1)))
(TPOP_<X{S}>                (MOVE () (X) (TSP +1)))
(TPOP_PC                    (MOVE () PC (TSP +1)))
(TPOP_X                     (MOVE () X (TSP +1)))
(TPOP_Y                     (MOVE () Y (TSP +1)))
(TPOP{V}_<Z>_Y              (MOVE (Z) Y (TSP +1)))
(TPOP{B}_<Z>_Y              (MOVE (Z) Y (TSP +1)))
(TPOP{I}_<ZN>_Y             (MOVE (Z N) Y (TSP +1)))
(TPOP{N}_<Z>_Y              (MOVE (Z) Y (TSP +1)))
(TPUSH_*                    (MOVE () (-1 TSP) (PC +1)))
(TPUSH_<X{A}>               (MOVE () (-1 TSP) (X)))
(TPUSH_<X{S}>               (MOVE () (-1 TSP) (X)))
(TPUSH_CSP                  (MOVE () (-1 TSP) CSP))
(TPUSH_TSP                  (MOVE () (-1 TSP) TSP))
(TPUSH_X                    (MOVE () (-1 TSP) X))
(XOR_<TSP>_<TSP>            (XOR () (TSP) (TSP)))
(XOR_<TSP>{V}_X{V}          (XOR () (TSP) X))
(XOR_<TSP>{B}_*{B}          (XOR () (TSP) (PC +1)))
(XOR_<TSP>{B}_X{B}          (XOR () (TSP) X))
(XOR_<Z>_<TSP>_X            (XOR (Z) (TSP) X)))
```

There are a total of 87 i-code opcodes. But some distinct i-code opcodes map to the same assembly language instruction. For example, both **XOR_<TSP>{V}_X{V}** and **XOR_<TSP>{B}_X{B}** map to **(XOR () (TSP) X)**. As is suggested by the annotations ‘‘**{V}**’’ and ‘‘**{B}**’’ in the opcode names, the first instruction deals with bit vectors and the second deals with Booleans. As manifested by the fact that they both map to a single instruction, no such distinction exists at the concrete level of FM8502. Why then do we have two different i-code instructions? The answer, for the moment, is that the type annotations serve the useful mnemonic role of helping us keep straight the types of objects we are manipulating. However, as we explain when we discuss the correctness proof, the annotations play a much deeper role: they let us factor the proof into a compiler proof and a link-assembler proof.

6.5.1.2. The Assembly Language

There are only 69 distinct assembly language instructions in the i-code table above. These could have been converted into the corresponding bit vectors by hand and listed in the table. However, it was less error-prone to implement a general purpose assembler for FM8502.

The structure of the assembly language should be pretty obvious from the examples above. The form of an assembly instruction is

```
(opcode c-codes operand-b operand-a),
```

where

- **`opcode`** determines the opcode of the FM8502 instruction and is any of following literal atoms **`INCR`**, **`ADDC`**, **`ADD`**, **`NEG`**, **`DECR`**, **`SUBB`**, **`SUB`**, **`ROR`**, **`ASR`**, **`LSR`**, **`XOR`**, **`OR`**, **`AND`**, **`NOT`**, **`MOVE`**, **`MOVE-NC`**, **`MOVE-C`**, **`MOVE-NV`**, **`MOVE-V`**, **`MOVE-NZ`**, **`MOVE-Z`**, **`MOVE-NN`** or **`MOVE-N`**;
- **`c-codes`** determines which of the condition code registers are set by the instruction and is a list containing any subset of {**`C`**, **`V`**, **`N`**, **`Z`**}; and
- **`operand-b`** and **`operand-a`** determine the two operands of the instruction and are each of one of the following forms: **`reg`**, **`(reg)`**, **`(-1 reg)`** or **`(reg +1)`** where **`reg`** is one of the following literal atoms: **`PC`**, **`CFP`**, **`CSP`**, **`TSP`**, **`X`** or **`Y`**.

The four different forms of operands describe both a register and an address mode. The address modes described are, respectively, register direct, register indirect, register indirect with pre-decrement, and register indirect with post-increment.

To assemble an FM8502 instruction, e.g., 32-bit wide bit vector, from an assembly instruction, the instruction assembler uses various tables to map the literal atoms above to numbers. For example, the **`LSR`** opcode is mapped to the number 10, which in binary is **`B10100`**, the FM8502 opcode/move bits for the ‘‘logical shift right, top bit zero’’ instruction. The condition codes and register names are similarly mapped to particular bit positions and register numbers. The FM8502 instruction is then assembled by arithmetic and finally converted to a bit vector. The details are given in the definition of **`MCI`**, page 198.

6.5.1.3. An Example

Consider the i-code instruction **`(TPUSH_X)`**. This instruction is mapped by the i-code table into the assembly instruction **`(MOVE () (-1 TSP) X)`**. This in turn is assembled into the natural number 1016420, which when converted to a bit vector in standard 32-bit binary notation is

```
B00000000000011111000001001100100.
```

Decoding this vector as an FM8502 instruction yields

```
opcode:           1111
move:                 1
i-bit:                 0
cvnz:                   0000
mode-b:                     10
reg-b:                        011
mode-a:                          00
reg-a:                             100
```

That is, the instruction is an unconditional (**cvnz: B0000**) move (**opcode/move: B11111**). Operand-a, the source of the move, is register 4 (**B100**) in register-direct mode (**B00**). Operand-b, the destination of the move, is register 3 (**B011**), in register-indirect with pre-decrement mode (**B10**). The effect of the instruction is to (a) increment the program counter, register 0, by 1; (b) fetch the contents, **x**, of register 4; (c) decrement the contents of register 3 by 1 and store the result in 3; (d) deposit **x** at the address contained in register 3. Observe that if register 3 contains a stack pointer and one uses the conventions that the stack pointer points to the topmost element of the stack and stacks grown downward, then this instruction pushes **x** onto the stack pointed to by register 3.

6.5.1.4. Use of the Addressing Modes

As the i-code table shows, the assembler makes extensive use of all four addressing modes. The effects achieved with the various combinations of modes is sometimes subtle. We explain our use of the modes here.

Pushes onto the stacks are achieved by using pre-decrement addressing mode on the stack pointer. Thus, as we saw above, **(TPUSH_X)**—which pushes the **x** register onto the temporary stack—is implemented by **(MOVE () (-1 TSP) X)**.

Pops are achieved by using post-increment addressing mode. Thus, **(TPOP_X)**—which pops the top of the temporary stack into the **x** register—is implemented with **(MOVE () X (TSP +1))**. The effect of this instruction is to (a) increment the program counter by 1; (b) fetch indirect through the address in the **TSP** register (obtaining the topmost element of the stack) ; (c) deposit the result into the **X** register; and (d) increment the **TSP** register by 1 (popping the stack).

When we move the actuals from the temporary stack to the control stack we can move each with one instruction: **(CPUSH_<TSP>+)** which maps to **(MOVE () (-1 CSP) (TSP +1))**. This instruction fetches the topmost element of **TSP**, pushes it onto **CSP**, and pops **TSP**.

Perhaps the most confusing addressing mode combination is when we use post-increment mode on the program counter. Consider the i-code instruction **(TPUSH_*)**. This instruction is supposed to push onto the temporary stack the next word in the instruction stream. Thus, the i-code sequence

```
(TPUSH_*)
(NAT 27)
(ADD_TSP_X{N})
```

will push 27 onto the temporary stack and then execute the **(ADD_TSP_X{N})** instruction. How do we fetch the next word in the instruction stream as data and how do we increment the program counter by 2 so that we do not also execute that word?

(TPUSH_*) is mapped into **(MOVE () (-1 TSP) (PC +1))**. The effect of this instruction is to (a) increment the program counter by 1, as usual; (b) fetch indirect through the **pc** register, which because of step (a) now points to the word after this instruction; (c) push the result onto the temporary stack; and (d) increment the **pc** register by 1, which makes it point to the instruction two words past the one being executed.

This completes our discussion of how i-code instructions are assembled into FM8502 machine code instructions. The details can be gleaned by reading the definition of **LINK-INSTR-WORD**, defined on page 195.

6.5.2. The Link Tables

We use four link tables. The first, called the *program link table*, maps each program name to the absolute location of the beginning of the program. The second table, called the *label tables*, maps each each program name to its ‘‘label table.’’ The *label table* for a program name maps the i-code labels to the absolute position of the definition of the label in the i-code program. We illustrate this in a moment. The third table, called the *user data link table*, maps each global data area name to the absolute location of the beginning of the associated array. The fourth table, called the *system data link table*, maps each of the five system data area names to the absolute location of the beginning of the associated area.

Here is a simple example. Below is a system of two Piton programs, **PTZ**, discussed above, and a main program that simply calls **PTZ**.

```
((MAIN NIL NIL
       (CALL PTZ)
       (RET))
 (PTZ  NIL NIL
  (DL  LOOP ()
       (TEST-NAT-AND-JUMP ZERO END))
       (JUMP LOOP)
  (DL  END  ()
       (RET))))
```

The i-code produced by compiling the above system is shown in Figure 6-7. We load these programs in the low part of memory, starting at address 0. Thus, the program link table produced from the i-code is

```
((MAIN .  0)
 (PTZ  . 11)).
```

That is, the **MAIN** program is loaded starting at absolute address **0** and the **PTZ** program is loaded immediately after the last instruction in **MAIN** and hence starts at absolute location **11**.

The label tables table for the i-code in Figure 6-7 is

```
((MAIN ((MAIN PRELUDE) .  0)
       ((MAIN . 0)     .  2)
       ((MAIN . 1)     .  6)
       ((MAIN . 2)     .  8))
 (PTZ  ((PTZ PRELUDE)  . 11)
       ((PTZ  . 0)     . 13)
       ((PTZ  . 1)     . 17)
       ((PTZ  . 2)     . 19)
       ((PTZ  . 3)     . 21))).
```

Figure 6-7: Compiler Output for MAIN and PTZ

```
                           i-code                  Piton
      label              instruction              instruction                      pc
((MAIN
  (DL (MAIN PRELUDE)                              (PRELUDE)
                   (CPUSH_CFP))                                                   ; 0
                   (MOVE_CFP_CSP)                                                 ; 1
  (DL (MAIN . 0)                                  (CALL PTZ)
                   (CPUSH_*))                                                     ; 2
                   (PC (MAIN . 1))                                                ; 3
                   (JUMP_*)                                                       ; 4
                   (PC (PTZ PRELUDE))                                             ; 5
  (DL (MAIN . 1)                                  (RET)
                   (JUMP_*))                                                      ; 6
                   (PC (MAIN . 2))                                                ; 7
  (DL (MAIN . 2)                                  (POSTLUDE)
                   (MOVE_CSP_CFP))                                                ; 8
                   (CPOP_CFP)                                                     ; 9
                   (CPOP_PC))                                                     ;10
 (PTZ
  (DL (PTZ PRELUDE)                               (PRELUDE)
                   (CPUSH_CFP))                                                   ; 0
                   (MOVE_CFP_CSP)                                                 ; 1
  (DL (PTZ . 0)                               (DL LOOP NIL
                                                  (TEST-NAT-AND-JUMP ZERO
                                                                   END))
                   (TPOP{N}_<Z>_Y))                                               ; 2
                   (MOVE_X_*)                                                     ; 3
                   (PC (PTZ . 2))                                                 ; 4
                   (JUMP-Z_X)                                                     ; 5
  (DL (PTZ . 1)                                   (JUMP LOOP)
                   (JUMP_*))                                                      ; 6
                   (PC (PTZ . 0))                                                 ; 7
  (DL (PTZ . 2)                               (DL END NIL (RET))
                   (JUMP_*))                                                      ; 8
                   (PC (PTZ . 3))                                                 ; 9
  (DL (PTZ . 3)                                   (POSTLUDE)
                   (MOVE_CSP_CFP))                                                ;10
                   (CPOP_CFP)                                                     ;11
                   (CPOP_PC)))                                                    ;12
```

For example, the i-code instruction to which the label **(MAIN . 1)** is attached will be loaded at absolute position **6**. That is, the absolute location defined by **(MAIN . 1)** is **6**.

Our convention is to load the data segment immediately after the program segment. Thus, if the data segment were

```
((A     (NAT 0)
        (NAT 1)
        (NAT 2)
        (NAT 3)
        (NAT 4))
 (B     (INT -2)
```

```
            (INT -1)
            (INT 0))
     (A-LEN (NAT 5))
     (B-LEN (NAT 3))),
```

then the user data link table would be

```
((A     . 24)
 (B     . 29)
 (A-LEN . 32)
 (B-LEN . 33)).
```

For example, the 0^{th} word in the array **B** is at absolute location **29**.

Finally, it is our convention to allocate the system data segment immediately after the user data segment. If the maximum control stack size were 10 and the maximum temporary stack size were 8, then the system data link table would be

```
((CSTK                . 34)
 (TSTK                . 45)
 (FULL-CTRL-STK-ADDR  . 54)
 (FULL-TEMP-STK-ADDR  . 55)
 (EMPTY-TEMP-STK-ADDR . 56)).
```

The computation of the four link tables is straightforward. See **I-LINK-TABLES**, page 181.

6.5.3. Linking Data Objects

Data objects are linked by the function **LINK-DATA-WORD** which is defined on page 195. Eight types of data objects are encountered by the linker: the seven Piton data types, **NAT**, **INT**, **BITV**, **BOOL**, **ADDR**, **PC** and **SUBR**, and the implementation-internal type **SYS-ADDR**. The linker uses the tag word to determine the type of the object and then maps it into a bit vector accordingly. We discuss each in turn.

6.5.3.1. Naturals

Natural numbers are represented in the standard binary notation. Thus, **(NAT 3)** is mapped to the 32-bit wide vector **B00000000000000000000000000000011**

6.5.3.2. Integers

We use the standard twos-complement representation of integers. Thus, **(INT 3)** maps to the same thing as **(NAT 3)**, above, and **(INT -3)** maps to **B11111111111111111111111111111101**.

6.5.3.3. Bit Vectors

Piton's bit vector objects are mapped directly to FM8502 bit vectors. Thus,

```
(BITV (0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
       0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1))
```

is mapped to **B00001111000011110000000011111111**.

6.5.3.4. Booleans

(BOOL F) is mapped to **B00000000000000000000000000000000** and **(BOOL T)** is mapped to **B00000000000000000000000000000001**.

6.5.3.5. Data Addresses

An object of the form **(ADDR (name . n))** is mapped first to a natural number and then to the bit vector representing that natural in binary notation. To map such a data address to a natural number we look in the data link table to get the base address of **name** and add **n** to it. For example, in the link tables shown above, **B** is assigned location **29**, so **(ADDR (B . 2))** maps to the bit vector representing 31, **B00000000000000000000000000011111**.

6.5.3.6. Program Addresses

An object of the form **(PC (name . x))** is mapped first to a natural number and then to the corresponding bit vector. The natural number is obtained by finding the label table for **name** in the label tables and then looking up **(name . x)** in that table. For example, given the example link tables shown above, **(PC (PTZ PRELUDE))** maps to the natural number 11 and thence to **B00000000000000000000000000001011**.

6.5.3.7. Subroutines

An object of the form **(SUBR name)** is mapped first to the natural number associated with **name** in the program link table and then to the corresponding bit vector.

6.5.3.8. System Data Addresses

An object of the form **(SYS-ADDR (name . n))** is mapped analogously to data addresses except that the base address of **name** is obtained from the system link table.

7. The Formal Definition of Piton

This chapter contains all of the formulas involved in the formal definition of Piton. The chapter is divided into two sections. The second section is simply a listing, in alphabetical order, of all of the definitions involved. These definitions are indexed and exhaustively cross-indexed in the Index of this report. The first section here is a guide to the second section. It briefly mentions each of the important ‘‘entry points’’ into the list of definitions.

7.1. A Guide to the Formal Definition of Piton

7.1.1. Proper P-States

P-states are formally represented by the **P-STATE** shell

Shell Definition.
Add the shell **P-STATE** of **9** arguments, with
recognizer function symbol **P-STATEP**, and
accessors **P-PC**, **P-CTRL-STK**, **P-TEMP-STK**, **P-PROG-SEGMENT**,
P-DATA-SEGMENT, **P-MAX-CTRL-STK-SIZE**, **P-MAX-TEMP-STK-SIZE**,
P-WORD-SIZE and **P-PSW**.

The proper p-states are described by **PROPER-P-STATEP**,

Definition.
```
(PROPER-P-STATEP P)
    =
(AND (P-STATEP P)                                                    ; (1)
     (P-OBJECTP-TYPE 'PC (P-PC P) P)                                 ; (2)
     (LISTP (P-CTRL-STK P))                                          ; (3)
     (PROPER-P-FRAMEP (TOP (P-CTRL-STK P))                           ; (4)
                      (AREA-NAME (P-PC P))
                      P)
     (PROPER-P-CTRL-STKP (POP (P-CTRL-STK P))                        ; (5)
                         (AREA-NAME (RET-PC (TOP (P-CTRL-STK P))))
                         P)
     (NOT (LESSP (P-MAX-CTRL-STK-SIZE P)                             ; (6)
                 (P-CTRL-STK-SIZE (P-CTRL-STK P))))
     (PROPER-P-TEMP-STKP (P-TEMP-STK P) P)                           ; (7)
     (NOT (LESSP (P-MAX-TEMP-STK-SIZE P)                             ; (8)
                 (LENGTH (P-TEMP-STK P))))
     (PROPER-P-PROG-SEGMENTP (P-PROG-SEGMENT P)                      ; (9)
                             P)
     (PROPER-P-DATA-SEGMENTP (P-DATA-SEGMENT P)                      ;(10)
                             P)
     (NUMBERP (P-MAX-CTRL-STK-SIZE P))                               ;(11)
     (NUMBERP (P-MAX-TEMP-STK-SIZE P))                               ;(12)
     (NUMBERP (P-WORD-SIZE P))                                       ;(13)
     (LESSP (P-MAX-CTRL-STK-SIZE P)                                  ;(14)
            (EXP 2 (P-WORD-SIZE P)))
     (LESSP (P-MAX-TEMP-STK-SIZE P)                                  ;(15)
            (EXP 2 (P-WORD-SIZE P)))
     (LESSP 0 (P-WORD-SIZE P))).                                     ;(16)
```

The sixteen conjuncts of this definition may be paraphrased as follows. (1) **P** is a p-state (see **P-STATE**, page 131); (2) the program counter of **P** is a legal Piton object of type **PC** (see **P-OBJECTP-TYPE**, page 122); (3) the control stack is non-empty; (4) the topmost frame of the control stack is a proper frame for the current program counter and state (see **PROPER-P-FRAMEP**, page 142); (5) the rest of the control stack is similarly proper (see **PROPER-P-CTRL-STKP**, page 141); (6) the ‘‘size’’ of the control stack does not exceed the specified limit (see **P-CTRL-STK-SIZE**, page 108); (7) the temporary stack is proper in the current state, which means it consists entirely of legal Piton objects (see **PROPER-P-TEMP-STKP**, page 150); (8) the length of the temporary stack does not exceed the specified limit; (9) the program segment is well-formed (see **PROPER-P-PROG-SEGMENTP**, page 147 and below); (10) the data segment is well-formed (see **PROPER-P-DATA-SEGMENTP**, page 142); (11-13) the maximum control stack size, the maximum temporary stack size, and the word size, w, are all natural numbers; (14) the maximum control stack size is less than 2^w; (15) the maximum temporary stack size is less than 2^w; and (16) the word-size is greater than 0.

The reader is encouraged to pursue each of the references above. We will briefly elaborate on the definition of **PROPER-P-PROG-SEGMENTP**.

Definition.

```
(PROPER-P-PROG-SEGMENTP SEGMENT P)
   =
(IF (NLISTP SEGMENT)
    (EQUAL SEGMENT NIL)
    (AND (PROPER-P-PROGRAMP (CAR SEGMENT) P)
         (PROPER-P-PROG-SEGMENTP (CDR SEGMENT)
                                 P)))
```

This function is recursive. It requires that the (program) **SEGMENT** on which it was called be a list ending in **NIL**, every element of which is a **PROPER-P-PROGRAMP** with respect to the current p-state.

Looking up **PROPER-P-PROGRAMP** in the index, we find that it is defined on page 147, as follows:

Definition.

```
(PROPER-P-PROGRAMP PROG P)
   =
(AND (LITATOM (NAME PROG))
     (ALL-LITATOMS (FORMAL-VARS PROG))
     (PROPER-P-TEMP-VAR-DCLSP (TEMP-VAR-DCLS PROG) P)
     (PROPER-P-PROGRAM-BODYP (PROGRAM-BODY PROG)
                             (NAME PROG)
                             P)).
```

This function checks that the name of the program is a literal atom, the formal variable field contains a list of literal atoms, the temporary variable declaration field contains a proper list of declarations (in particular, each specifies a literal atom as a variable and a legal Piton object as its initial value), and the body of the program is proper.

On page 147 we see

Definition.

```
(PROPER-P-PROGRAM-BODYP LST NAME P)
   =
(AND (LISTP LST)
     (PROPER-LABELED-P-INSTRUCTIONSP LST NAME P)
     (FALL-OFF-PROOFP LST)).
```

That is, a proper Piton program body is a non-empty list of properly labeled Piton instructions that is ‘‘fall off proof.’’ The latter term means that the last instruction in the list is an unconditional transfer of control—i.e., it is not possible to ‘‘fall off the end’’ of the list by executing the instructions.

We continue our dive by looking at **PROPER-LABELED-P-INSTRUCTIONSP**

Definition.

```
(PROPER-LABELED-P-INSTRUCTIONSP LST NAME P)
   =
(IF (NLISTP LST)
    (EQUAL LST NIL)
    (AND (LEGAL-LABELP (CAR LST))
         (PROPER-P-INSTRUCTIONP (UNLABEL (CAR LST)) NAME P)
         (PROPER-LABELED-P-INSTRUCTIONSP (CDR LST) NAME P))).
```

This recursive function checks that each element of the body is legally labelled (if at all) and that the result of unlabelling the element is a **PROPER-P-INSTRUCTIONP** as defined on page 143

Definition.

```
(PROPER-P-INSTRUCTIONP INS NAME P)
   =
(AND
 (PROPERP INS)
 (CASE
  (CAR INS)
  (CALL          (PROPER-P-CALL-INSTRUCTIONP INS NAME P))
  (RET           (PROPER-P-RET-INSTRUCTIONP INS NAME P))
  (LOCN          (PROPER-P-LOCN-INSTRUCTIONP INS NAME P))
  (PUSH-CONSTANT (PROPER-P-PUSH-CONSTANT-INSTRUCTIONP INS NAME P))
  ...
  (OR-BOOL       (PROPER-P-OR-BOOL-INSTRUCTIONP INS NAME P))
  (AND-BOOL      (PROPER-P-AND-BOOL-INSTRUCTIONP INS NAME P))
  (NOT-BOOL      (PROPER-P-NOT-BOOL-INSTRUCTIONP INS NAME P))
  (OTHERWISE     F))).
```

The **CASE** construct is an abbreviation for an **IF** nest. **(CASE x (key val) ...)** abbreviates **(IF (EQUAL x ’key) val (CASE x ...))** and **(CASE x (OTHERWISE val))** abbreviates **val**. Thus, **PROPER-P-INSTRUCTIONP** splits on the opcode of the Piton instruction and for each opcode calls the appropriate predicate to check that the instruction is well-formed.

Consider, for example, the **PUSH-CONSTANT** instruction. The predicate that checks whether it is well-formed is named **PROPER-P-PUSH-CONSTANT-INSTRUCTIONP**

Definition.

```
(PROPER-P-PUSH-CONSTANT-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 2)
     (OR (P-OBJECTP (CADR INS) P)
         (EQUAL (CADR INS) ’PC)
         (FIND-LABELP (CADR INS)
                      (PROGRAM-BODY (DEFINITION NAME
                                                (P-PROG-SEGMENT P))))))).
```

This predicate checks that the instruction is of length 2, i.e., has the form **(PUSH-CONSTANT x)**, and that **x** is either a legal Piton object in the state containing the instruction, or is the atom **’PC**, or is a label in the program containing the instruction.

In general, to determine the syntactic restrictions on an instruction named **opcode**, look at the definition of the function named **PROPER-P-opcode-INSTRUCTIONP**.

This completes our brief tour through the definition of proper p-states. The serious student of Piton should use the index to trace out the entire tree of definitions.

7.1.2. The Piton Interpreter

The Piton interpreter is the function named **P**.

Definition.
```
(P P N)
   =
(IF (ZEROP N)
    P
    (P (P-STEP P) (SUB1 N)))
```
This function steps the proper p-state **P** a specified number of times, **N**.

The single-stepper for Piton is

Definition.
```
(P-STEP P)
   =
(IF (EQUAL (P-PSW P) 'RUN)
    (P-STEP1 (P-CURRENT-INSTRUCTION P) P)
    P).
```
Note that **P-STEP** is a no-op if the psw is not **'RUN**. If the psw is **'RUN** we use the function **P-STEP1** on the current instruction and the current p-state.

Definition.
```
(P-STEP1 INS P)
   =
(IF (P-INS-OKP INS P)
    (P-INS-STEP INS P)
    (P-HALT P (X-Y-ERROR-MSG 'P (CAR INS))))
```
P-STEP1 first checks the precondition of the current instruction, using **P-INS-OKP**. If the precondition is satisfied, **P-INS-STEP** is used to compute the next state. Otherwise, the psw of the current state is set to an error message.

The two functions, **P-INS-OKP** and **P-INS-STEP** are defined as **CASE** splits on the opcode of the current instruction. For each opcode there is a function that checks the precondition and another that computes the step. To determine the precondition of an the instruction **opcode** look up the function **P-opcode-OKP**. To determine the step function for that instruction, see **P-opcode-STEP**.

Consider the **PUSH-CONSTANT** instruction again. The precondition for that instruction is encoded in

Definition.
```
(P-PUSH-CONSTANT-OKP INS P)
   =
(LESSP (LENGTH (P-TEMP-STK P))
       (P-MAX-TEMP-STK-SIZE P)).
```
This function merely checks that there is room on the temporary stack to do one more push. The syntactic constraints on proper p-states assures us that the object to be pushed is a legal Piton object (given our

treatment of the special token **PC** and of labels). The syntactic constraints also assure us that it is legal to increment the program counter by one (it is impossible to fall off the end of a proper Piton program).

The step function for **PUSH-CONSTANT** is

Definition.
```
(P-PUSH-CONSTANT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (UNABBREVIATE-CONSTANT (CADR INS) P)
               (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN).
```

Note that the step function increments the program counter by one, pushes one thing onto the temporary stack (obtained by ‘‘unabbreviating’’ the operand of the instruction, and does not alter any other component of the current state.

The precondition and effects of all other Piton instructions are defined similarly. The formal definitions below (accessed via the Index or via the **P-opcode-OKP**/**P-opcode-STEP** naming convention) are offered as a precise reference manual for Piton.

Readers uninterested in pursuing the formal definition at this time should skip to page 155.

7.2. Alphabetical Listing of the Piton Definition

Definition.
```
(ADD-ADDR ADDR N)
   =
(TAG (TYPE ADDR)
     (ADD-ADP (UNTAG ADDR) N))
```

Definition.
```
(ADD-ADP ADP N)
   =
(CONS (ADP-NAME ADP)
      (PLUS (ADP-OFFSET ADP) N))
```

Definition.
```
(ADD1-ADDR ADDR)
   =
(ADD-ADDR ADDR 1)
```

Definition.
```
(ADD1-P-PC P)
   =
(ADD1-ADDR (P-PC P))
```

Definition.
```
(ADP-NAME ADP)
   =
(CAR ADP)
```

Definition.
```
(ADP-OFFSET ADP)
   =
(CDR ADP)
```

Definition.
```
(ADPP X SEGMENT)
   =
(AND (LISTP X)
     (NUMBERP (ADP-OFFSET X))
     (DEFINEDP (ADP-NAME X) SEGMENT)
     (LESSP (ADP-OFFSET X)
            (LENGTH (VALUE (ADP-NAME X) SEGMENT))))
```

Definition.
```
(ALL-BUT-LAST A)
   =
(COND ((NLISTP A) NIL)
      ((NLISTP (CDR A)) NIL)
      (T (CONS (CAR A)
               (ALL-BUT-LAST (CDR A)))))
```

Definition.
```
(ALL-FIND-LABELP LAB-LST LST)
   =
(IF (NLISTP LAB-LST)
    T
    (AND (FIND-LABELP (CAR LAB-LST) LST)
         (ALL-FIND-LABELP (CDR LAB-LST) LST)))
```

Definition.
```
(ALL-LITATOMS LST)
   =
(IF (NLISTP LST)
    (EQUAL LST NIL)
    (AND (LITATOM (CAR LST))
         (ALL-LITATOMS (CDR LST))))
```

Definition.
```
(ALL-P-OBJECTPS LST P)
   =
(IF (NLISTP LST)
    (EQUAL LST NIL)
    (AND (P-OBJECTP (CAR LST) P)
         (ALL-P-OBJECTPS (CDR LST) P)))
```

Definition.
```
(ALL-ZERO-BITVP A)
   =
(IF (LISTP A)
    (AND (EQUAL (CAR A) 0)
         (ALL-ZERO-BITVP (CDR A)))
    T)
```

Definition.
```
(AND-BIT BIT1 BIT2)
   =
(COND ((EQUAL BIT1 0) 0)
      ((EQUAL BIT2 0) 0)
      (T 1))
```

Definition.
```
(AND-BITV A B)
   =
(IF (NLISTP A)
    NIL
    (CONS (AND-BIT (CAR A) (CAR B))
          (AND-BITV (CDR A) (CDR B))))
```

Definition.
```
(AND-BOOL X Y)
   =
(IF (EQUAL X 'F) 'F Y)
```

Definition.
```
(AREA-NAME X)
   =
(ADP-NAME (UNTAG X))
```

Definition.
```
(BINDINGS FRAME)
   =
(CAR FRAME)
```

Definition.
```
(BIT-VECTORP X N)
   =
(IF (NLISTP X)
    (AND (EQUAL X NIL) (ZEROP N))
    (AND (NOT (ZEROP N))
         (BITP (CAR X))
         (BIT-VECTORP (CDR X) (SUB1 N))))
```

Definition.
```
(BITP X)
   =
(OR (EQUAL X 0) (EQUAL X 1))
```

Definition.
```
(BOOL X)
   =
(TAG 'BOOL (IF X 'T 'F))
```

Definition.
```
(BOOL-TO-NAT FLG)
   =
(IF (EQUAL FLG 'F) 0 1)
```

Definition.
```
(BOOLEANP X)
   =
(OR (EQUAL X 'T) (EQUAL X 'F))
```

Definition.
```
(DEFINEDP NAME ALIST)
   =
(COND ((NLISTP ALIST) F)
      ((EQUAL NAME (CAAR ALIST)) T)
      (T (DEFINEDP NAME (CDR ALIST))))
```

Definition.
```
(DEFINITION NAME ALIST)
   =
(ASSOC NAME ALIST)
```

Definition.
```
(DEPOSIT VAL ADDR SEGMENT)
   =
(DEPOSIT-ADP VAL (UNTAG ADDR) SEGMENT)
```

Definition.
```
(DEPOSIT-ADP VAL ADP SEGMENT)
   =
(PUT-VALUE (PUT VAL
                (ADP-OFFSET ADP)
                (VALUE (ADP-NAME ADP) SEGMENT))
           (ADP-NAME ADP)
           SEGMENT)
```

Definition.
```
(EXP I J)
   =
(IF (ZEROP J)
    1
    (TIMES I (EXP I (SUB1 J))))
```

Definition.
```
(FALL-OFF-PROOFP LST)
   =
(MEMBER (CAR (UNLABEL (GET (SUB1 (LENGTH LST)) LST)))
        '(RET JUMP JUMP-CASE POPJ))
```

Definition.
```
(FETCH ADDR SEGMENT)
   =
(FETCH-ADP (UNTAG ADDR) SEGMENT)
```

Definition.
```
(FETCH-ADP ADP SEGMENT)
   =
(GET (ADP-OFFSET ADP)
     (VALUE (ADP-NAME ADP) SEGMENT))
```

Definition.
```
(FIND-LABEL X LST)
   =
(COND ((NLISTP LST) 0)
      ((AND (LABELLEDP (CAR LST))
            (EQUAL X (CADAR LST)))
       0)
      (T (ADD1 (FIND-LABEL X (CDR LST)))))
```

Definition.
```
(FIND-LABELP X LST)
   =
(COND ((NLISTP LST) F)
      ((AND (LABELLEDP (CAR LST))
            (EQUAL X (CADAR LST)))
       T)
      (T (FIND-LABELP X (CDR LST))))
```

Definition.
```
(FIRST-N N X)
   =
(IF (ZEROP N)
    NIL
    (CONS (CAR X)
          (FIRST-N (SUB1 N) (CDR X))))
```

Definition.
```
(FIX-SMALL-INTEGER I WORD-SIZE)
   =
(COND ((SMALL-INTEGERP I WORD-SIZE) I)
      ((NEGATIVEP I)
       (IPLUS I (EXP 2 WORD-SIZE)))
      (T (IPLUS I (MINUS (EXP 2 WORD-SIZE)))))
```

Definition.
```
(FIX-SMALL-NATURAL N WORD-SIZE)
   =
(REMAINDER N (EXP 2 WORD-SIZE))
```

Definition.
```
(FORMAL-VARS D)
   =
(CADR D)
```

Definition.
```
(GET N LST)
   =
(IF (ZEROP N)
    (CAR LST)
    (GET (SUB1 N) (CDR LST)))
```

Definition.
```
(IDIFFERENCE I J)
   =
(IPLUS I (INEGATE J))
```

Definition.
```
(ILESSP I J)
   =
(COND ((NEGATIVEP I)
       (IF (NEGATIVEP J)
           (LESSP (NEGATIVE-GUTS J)
                  (NEGATIVE-GUTS I))
           T))
      ((NEGATIVEP J) F)
      (T (LESSP I J)))
```

Definition.
```
(INEGATE I)
   =
(COND ((NEGATIVEP I) (NEGATIVE-GUTS I))
      ((ZEROP I) 0)
      (T (MINUS I)))
```

Definition.
```
(INTEGERP I)
   =
(OR (NUMBERP I)
    (AND (NEGATIVEP I)
         (NOT (EQUAL (NEGATIVE-GUTS I) 0))))
```

Definition.
```
(IPLUS I J)
   =
(COND ((NEGATIVEP I)
       (COND ((NEGATIVEP J)
              (MINUS (PLUS (NEGATIVE-GUTS I)
                           (NEGATIVE-GUTS J))))
             ((LESSP J (NEGATIVE-GUTS I))
              (MINUS (DIFFERENCE (NEGATIVE-GUTS I) J)))
             (T (DIFFERENCE J (NEGATIVE-GUTS I)))))
      ((NEGATIVEP J)
       (IF (LESSP I (NEGATIVE-GUTS J))
           (MINUS (DIFFERENCE (NEGATIVE-GUTS J) I))
           (DIFFERENCE I (NEGATIVE-GUTS J))))
      (T (PLUS I J)))
```

Definition.
```
(LABELLEDP X)
   =
(EQUAL (CAR X) 'DL)
```

Definition.
```
(LEGAL-LABELP INS)
   =
(IMPLIES (LABELLEDP INS)
         (LITATOM (CADR INS)))
```

Definition.
```
(LENGTH X)
   =
(IF (NLISTP X)
    0
    (ADD1 (LENGTH (CDR X))))
```

Definition.
```
(LOCAL-VAR-VALUE VAR CTRL-STK)
   =
(VALUE VAR (BINDINGS (TOP CTRL-STK)))
```

Definition.
```
(LOCAL-VARS D)
   =
(APPEND (FORMAL-VARS D)
        (STRIP-CARS (TEMP-VAR-DCLS D)))
```

Definition.
```
(LSH-BITV A)
   =
(APPEND (CDR A) '(0))
```

Definition.
```
(MAKE-P-CALL-FRAME FORMAL-VARS TEMP-STK TEMP-VAR-DCLS RET-PC)
   =
(P-FRAME (APPEND (PAIR-FORMAL-VARS-WITH-ACTUALS FORMAL-VARS
                                                TEMP-STK)
                 (PAIR-TEMPS-WITH-INITIAL-VALUES TEMP-VAR-DCLS))
         RET-PC)
```

Definition.
```
(NAME D)
   =
(CAR D)
```

Definition.
```
(NOT-BIT BIT)
   =
(IF (EQUAL BIT 0) 1 0)
```

Definition.
```
(NOT-BITV A)
   =
(IF (NLISTP A)
    NIL
    (CONS (NOT-BIT (CAR A))
          (NOT-BITV (CDR A))))
```

Definition.
```
(NOT-BOOL X)
   =
(IF (EQUAL X 'F) 'T 'F)
```

Definition.
```
(OFFSET X)
   =
(ADP-OFFSET (UNTAG X))
```

Definition.
```
(OR-BIT BIT1 BIT2)
   =
(IF (EQUAL BIT1 0)
    (IF (EQUAL BIT2 0) 0 1)
    1)
```

Definition.
```
(OR-BITV A B)
   =
(IF (NLISTP A)
    NIL
    (CONS (OR-BIT (CAR A) (CAR B))
          (OR-BITV (CDR A) (CDR B))))
```

Definition.
```
(OR-BOOL X Y)
   =
(IF (EQUAL X 'F) Y 'T)
```

Definition.
```
(P P N)
   =
(IF (ZEROP N)
    P
    (P (P-STEP P) (SUB1 N)))
```

Definition.
```
(P-ADD-ADDR-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'NAT
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'ADDR
                     (TOP1 (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'ADDR
                     (ADD-ADDR (TOP1 (P-TEMP-STK P))
                               (UNTAG (TOP (P-TEMP-STK P))))
                     P))
```

Definition.
```
(P-ADD-ADDR-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (ADD-ADDR (TOP1 (P-TEMP-STK P))
                         (UNTAG (TOP (P-TEMP-STK P))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-ADD-INT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'INT
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'INT
                     (TOP1 (P-TEMP-STK P))
                     P)
     (SMALL-INTEGERP (IPLUS (UNTAG (TOP1 (P-TEMP-STK P)))
                            (UNTAG (TOP (P-TEMP-STK P))))
                     (P-WORD-SIZE P)))
```

Definition.
```
(P-ADD-INT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'INT
                    (IPLUS (UNTAG (TOP1 (P-TEMP-STK P)))
                           (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-ADD-INT-WITH-CARRY-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (LISTP (POP (POP (P-TEMP-STK P))))
     (P-OBJECTP-TYPE 'INT
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'INT
                     (TOP1 (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'BOOL
                     (TOP2 (P-TEMP-STK P))
                     P))
```

Definition.
```
(P-ADD-INT-WITH-CARRY-STEP INS P)
   =
(P-STATE
 (ADD1-P-PC P)
 (P-CTRL-STK P)
 (PUSH
  (TAG 'INT
       (FIX-SMALL-INTEGER
        (IPLUS (BOOL-TO-NAT (UNTAG (TOP2 (P-TEMP-STK P))))
               (IPLUS (UNTAG (TOP1 (P-TEMP-STK P)))
                      (UNTAG (TOP (P-TEMP-STK P)))))
        (P-WORD-SIZE P)))
  (PUSH
   (BOOL (NOT (SMALL-INTEGERP
               (IPLUS (BOOL-TO-NAT (UNTAG (TOP2 (P-TEMP-STK P))))
                      (IPLUS (UNTAG (TOP1 (P-TEMP-STK P)))
                             (UNTAG (TOP (P-TEMP-STK P)))))
               (P-WORD-SIZE P))))
   (POP (POP (POP (P-TEMP-STK P))))))
 (P-PROG-SEGMENT P)
 (P-DATA-SEGMENT P)
 (P-MAX-CTRL-STK-SIZE P)
 (P-MAX-TEMP-STK-SIZE P)
 (P-WORD-SIZE P)
 'RUN)
```

Definition .

```
(P-ADD-NAT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'NAT
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'NAT
                     (TOP1 (P-TEMP-STK P))
                     P)
     (SMALL-NATURALP (PLUS (UNTAG (TOP1 (P-TEMP-STK P)))
                           (UNTAG (TOP (P-TEMP-STK P))))
                     (P-WORD-SIZE P)))
```

Definition .

```
(P-ADD-NAT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'NAT
                    (PLUS (UNTAG (TOP1 (P-TEMP-STK P)))
                          (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition .

```
(P-ADD-NAT-WITH-CARRY-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (LISTP (POP (POP (P-TEMP-STK P))))
     (P-OBJECTP-TYPE 'NAT
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'NAT
                     (TOP1 (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'BOOL
                     (TOP2 (P-TEMP-STK P))
                     P))
```

Definition.
```
(P-ADD-NAT-WITH-CARRY-STEP INS P)
   =
(P-STATE
 (ADD1-P-PC P)
 (P-CTRL-STK P)
 (PUSH
  (TAG 'NAT
       (FIX-SMALL-NATURAL
        (PLUS (BOOL-TO-NAT (UNTAG (TOP2 (P-TEMP-STK P))))
              (UNTAG (TOP1 (P-TEMP-STK P)))
              (UNTAG (TOP (P-TEMP-STK P))))
        (P-WORD-SIZE P)))
  (PUSH
   (BOOL (NOT (SMALL-NATURALP
               (PLUS (BOOL-TO-NAT (UNTAG (TOP2 (P-TEMP-STK P))))
                     (UNTAG (TOP1 (P-TEMP-STK P)))
                     (UNTAG (TOP (P-TEMP-STK P))))
               (P-WORD-SIZE P))))
   (POP (POP (POP (P-TEMP-STK P))))))
 (P-PROG-SEGMENT P)
 (P-DATA-SEGMENT P)
 (P-MAX-CTRL-STK-SIZE P)
 (P-MAX-TEMP-STK-SIZE P)
 (P-WORD-SIZE P)
 'RUN)
```

Definition.
```
(P-ADD1-INT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'INT
                     (TOP (P-TEMP-STK P))
                     P)
     (SMALL-INTEGERP (IPLUS 1 (UNTAG (TOP (P-TEMP-STK P))))
                     (P-WORD-SIZE P)))
```

Definition.
```
(P-ADD1-INT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'INT
                    (IPLUS 1
                           (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-ADD1-NAT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'NAT
                     (TOP (P-TEMP-STK P))
                     P)
     (SMALL-NATURALP (ADD1 (UNTAG (TOP (P-TEMP-STK P))))
                     (P-WORD-SIZE P)))
```

Definition.
```
(P-ADD1-NAT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'NAT
                    (ADD1 (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-AND-BITV-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'BITV
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'BITV
                     (TOP1 (P-TEMP-STK P))
                     P))
```

Definition.
```
(P-AND-BITV-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'BITV
                    (AND-BITV (UNTAG (TOP1 (P-TEMP-STK P)))
                              (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-AND-BOOL-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'BOOL
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'BOOL
                     (TOP1 (P-TEMP-STK P))
                     P))
```

Definition.
```
(P-AND-BOOL-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'BOOL
                    (AND-BOOL (UNTAG (TOP1 (P-TEMP-STK P)))
                              (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-CALL-OKP INS P)
   =
(AND
 (NOT
  (LESSP
   (P-MAX-CTRL-STK-SIZE P)
   (P-CTRL-STK-SIZE
    (PUSH (MAKE-P-CALL-FRAME (FORMAL-VARS
                              (DEFINITION (CADR INS)
                                          (P-PROG-SEGMENT P)))
                             (P-TEMP-STK P)
                             (TEMP-VAR-DCLS
                              (DEFINITION (CADR INS)
                                          (P-PROG-SEGMENT P)))
                             (ADD1-ADDR (P-PC P)))
          (P-CTRL-STK P)))))
 (NOT (LESSP (LENGTH (P-TEMP-STK P))
             (LENGTH (FORMAL-VARS
                      (DEFINITION (CADR INS)
                                  (P-PROG-SEGMENT P)))))))
```

Definition.
```
(P-CALL-STEP INS P)
   =
(P-STATE
 (TAG 'PC (CONS (CADR INS) 0))
 (PUSH (MAKE-P-CALL-FRAME (FORMAL-VARS
                            (DEFINITION (CADR INS)
                                        (P-PROG-SEGMENT P)))
                          (P-TEMP-STK P)
                          (TEMP-VAR-DCLS
                            (DEFINITION (CADR INS)
                                        (P-PROG-SEGMENT P)))
                          (ADD1-ADDR (P-PC P)))
       (P-CTRL-STK P))
 (POPN (LENGTH (FORMAL-VARS
                (DEFINITION (CADR INS)
                            (P-PROG-SEGMENT P))))
       (P-TEMP-STK P))
 (P-PROG-SEGMENT P)
 (P-DATA-SEGMENT P)
 (P-MAX-CTRL-STK-SIZE P)
 (P-MAX-TEMP-STK-SIZE P)
 (P-WORD-SIZE P)
 'RUN)
```

Definition.
```
(P-CTRL-STK-SIZE CTRL-STK)
   =
(IF (NLISTP CTRL-STK)
    0
    (PLUS (P-FRAME-SIZE (TOP CTRL-STK))
          (P-CTRL-STK-SIZE (CDR CTRL-STK))))
```

Definition.
```
(P-CURRENT-INSTRUCTION P)
   =
(UNLABEL (GET (OFFSET (P-PC P))
              (PROGRAM-BODY (P-CURRENT-PROGRAM P))))
```

Definition.
```
(P-CURRENT-PROGRAM P)
   =
(DEFINITION (AREA-NAME (P-PC P))
            (P-PROG-SEGMENT P))
```

Definition.
```
(P-DEPOSIT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'ADDR
                     (TOP (P-TEMP-STK P))
                     P))
```

Definition.

```
(P-DEPOSIT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (POP (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (DEPOSIT (TOP1 (P-TEMP-STK P))
                  (TOP (P-TEMP-STK P))
                  (P-DATA-SEGMENT P))
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.

```
(P-DEPOSIT-TEMP-STK-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'NAT
                     (TOP (P-TEMP-STK P))
                     P)
     (LESSP (UNTAG (TOP (P-TEMP-STK P)))
            (LENGTH (POP (POP (P-TEMP-STK P))))))
```

Definition.

```
(P-DEPOSIT-TEMP-STK-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (RPUT (TOP1 (P-TEMP-STK P))
               (UNTAG (TOP (P-TEMP-STK P)))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.

```
(P-DIV2-NAT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'NAT
                     (TOP (P-TEMP-STK P))
                     P)
     (LESSP (LENGTH (P-TEMP-STK P))
            (P-MAX-TEMP-STK-SIZE P)))
```

Definition.
```
(P-DIV2-NAT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'NAT
                    (REMAINDER (UNTAG (TOP (P-TEMP-STK P)))
                               2))
               (PUSH (TAG 'NAT
                          (QUOTIENT (UNTAG (TOP (P-TEMP-STK P)))
                                    2))
                     (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-EQ-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (EQUAL (TYPE (TOP (P-TEMP-STK P)))
            (TYPE (TOP1 (P-TEMP-STK P)))))
```

Definition.
```
(P-EQ-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (BOOL (EQUAL (UNTAG (TOP1 (P-TEMP-STK P)))
                            (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-FETCH-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'ADDR
                     (TOP (P-TEMP-STK P))
                     P))
```

Definition.

```
(P-FETCH-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (FETCH (TOP (P-TEMP-STK P))
                      (P-DATA-SEGMENT P))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.

```
(P-FETCH-TEMP-STK-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'NAT
                     (TOP (P-TEMP-STK P))
                     P)
     (LESSP (UNTAG (TOP (P-TEMP-STK P)))
            (LENGTH (P-TEMP-STK P))))
```

Definition.

```
(P-FETCH-TEMP-STK-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (RGET (UNTAG (TOP (P-TEMP-STK P)))
                     (P-TEMP-STK P))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.

```
(P-FRAME BINDINGS RET-PC)
   =
(LIST BINDINGS RET-PC)
```

Definition.

```
(P-FRAME-SIZE FRAME)
   =
(PLUS 2 (LENGTH (BINDINGS FRAME)))
```

Definition.

```
(P-HALT P PSW)
   =
(P-STATE (P-PC P)
         (P-CTRL-STK P)
         (P-TEMP-STK P)
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         PSW)
```

Definition.

```
(P-INS-OKP INS P)
   =
(CASE (CAR INS)
      (CALL                        (P-CALL-OKP INS P))
      (RET                         (P-RET-OKP INS P))
      (LOCN                        (P-LOCN-OKP INS P))
      (PUSH-CONSTANT               (P-PUSH-CONSTANT-OKP INS P))
      (PUSH-LOCAL                  (P-PUSH-LOCAL-OKP INS P))
      (PUSH-GLOBAL                 (P-PUSH-GLOBAL-OKP INS P))
      (PUSH-CTRL-STK-FREE-SIZE     (P-PUSH-CTRL-STK-FREE-SIZE-OKP INS P))
      (PUSH-TEMP-STK-FREE-SIZE     (P-PUSH-TEMP-STK-FREE-SIZE-OKP INS P))
      (PUSH-TEMP-STK-INDEX         (P-PUSH-TEMP-STK-INDEX-OKP INS P))
      (JUMP-IF-TEMP-STK-FULL       (P-JUMP-IF-TEMP-STK-FULL-OKP INS P))
      (JUMP-IF-TEMP-STK-EMPTY      (P-JUMP-IF-TEMP-STK-EMPTY-OKP INS P))
      (POP                         (P-POP-OKP INS P))
      (POP*                        (P-POP*-OKP INS P))
      (POPN                        (P-POPN-OKP INS P))
      (POP-LOCAL                   (P-POP-LOCAL-OKP INS P))
      (POP-GLOBAL                  (P-POP-GLOBAL-OKP INS P))
      (POP-LOCN                    (P-POP-LOCN-OKP INS P))
      (POP-CALL                    (P-POP-CALL-OKP INS P))
      (FETCH-TEMP-STK              (P-FETCH-TEMP-STK-OKP INS P))
      (DEPOSIT-TEMP-STK            (P-DEPOSIT-TEMP-STK-OKP INS P))
      (JUMP                        (P-JUMP-OKP INS P))
      (JUMP-CASE                   (P-JUMP-CASE-OKP INS P))
      (PUSHJ                       (P-PUSHJ-OKP INS P))
      (POPJ                        (P-POPJ-OKP INS P))
      (SET-LOCAL                   (P-SET-LOCAL-OKP INS P))
      (SET-GLOBAL                  (P-SET-GLOBAL-OKP INS P))
      (TEST-NAT-AND-JUMP           (P-TEST-NAT-AND-JUMP-OKP INS P))
      (TEST-INT-AND-JUMP           (P-TEST-INT-AND-JUMP-OKP INS P))
      (TEST-BOOL-AND-JUMP          (P-TEST-BOOL-AND-JUMP-OKP INS P))
      (TEST-BITV-AND-JUMP          (P-TEST-BITV-AND-JUMP-OKP INS P))
      (NO-OP                       (P-NO-OP-OKP INS P))
      (ADD-ADDR                    (P-ADD-ADDR-OKP INS P))
      (SUB-ADDR                    (P-SUB-ADDR-OKP INS P))
      (EQ                          (P-EQ-OKP INS P))
      (LT-ADDR                     (P-LT-ADDR-OKP INS P))
      (FETCH                       (P-FETCH-OKP INS P))
      (DEPOSIT                     (P-DEPOSIT-OKP INS P))
      (ADD-INT                     (P-ADD-INT-OKP INS P))
      (ADD-INT-WITH-CARRY          (P-ADD-INT-WITH-CARRY-OKP INS P))
      (ADD1-INT                    (P-ADD1-INT-OKP INS P))
      (SUB-INT                     (P-SUB-INT-OKP INS P))
```

```
      (SUB-INT-WITH-CARRY         (P-SUB-INT-WITH-CARRY-OKP INS P))
      (SUB1-INT                   (P-SUB1-INT-OKP INS P))
      (NEG-INT                    (P-NEG-INT-OKP INS P))
      (LT-INT                     (P-LT-INT-OKP INS P))
      (INT-TO-NAT                 (P-INT-TO-NAT-OKP INS P))
      (ADD-NAT                    (P-ADD-NAT-OKP INS P))
      (ADD-NAT-WITH-CARRY         (P-ADD-NAT-WITH-CARRY-OKP INS P))
      (ADD1-NAT                   (P-ADD1-NAT-OKP INS P))
      (SUB-NAT                    (P-SUB-NAT-OKP INS P))
      (SUB-NAT-WITH-CARRY         (P-SUB-NAT-WITH-CARRY-OKP INS P))
      (SUB1-NAT                   (P-SUB1-NAT-OKP INS P))
      (LT-NAT                     (P-LT-NAT-OKP INS P))
      (MULT2-NAT                  (P-MULT2-NAT-OKP INS P))
      (MULT2-NAT-WITH-CARRY-OUT   (P-MULT2-NAT-WITH-CARRY-OUT-OKP INS P))
      (DIV2-NAT                   (P-DIV2-NAT-OKP INS P))
      (OR-BITV                    (P-OR-BITV-OKP INS P))
      (AND-BITV                   (P-AND-BITV-OKP INS P))
      (NOT-BITV                   (P-NOT-BITV-OKP INS P))
      (XOR-BITV                   (P-XOR-BITV-OKP INS P))
      (RSH-BITV                   (P-RSH-BITV-OKP INS P))
      (LSH-BITV                   (P-LSH-BITV-OKP INS P))
      (OR-BOOL                    (P-OR-BOOL-OKP INS P))
      (AND-BOOL                   (P-AND-BOOL-OKP INS P))
      (NOT-BOOL                   (P-NOT-BOOL-OKP INS P))
      (OTHERWISE                  F))
```

Definition.

```
(P-INS-STEP INS P)
  =
(CASE (CAR INS)
      (CALL                      (P-CALL-STEP INS P))
      (RET                       (P-RET-STEP INS P))
      (LOCN                      (P-LOCN-STEP INS P))
      (PUSH-CONSTANT             (P-PUSH-CONSTANT-STEP INS P))
      (PUSH-LOCAL                (P-PUSH-LOCAL-STEP INS P))
      (PUSH-GLOBAL               (P-PUSH-GLOBAL-STEP INS P))
      (PUSH-CTRL-STK-FREE-SIZE   (P-PUSH-CTRL-STK-FREE-SIZE-STEP INS P))
      (PUSH-TEMP-STK-FREE-SIZE   (P-PUSH-TEMP-STK-FREE-SIZE-STEP INS P))
      (PUSH-TEMP-STK-INDEX       (P-PUSH-TEMP-STK-INDEX-STEP INS P))
      (JUMP-IF-TEMP-STK-FULL     (P-JUMP-IF-TEMP-STK-FULL-STEP INS P))
      (JUMP-IF-TEMP-STK-EMPTY    (P-JUMP-IF-TEMP-STK-EMPTY-STEP INS P))
      (POP                       (P-POP-STEP INS P))
      (POP*                      (P-POP*-STEP INS P))
      (POPN                      (P-POPN-STEP INS P))
      (POP-LOCAL                 (P-POP-LOCAL-STEP INS P))
      (POP-GLOBAL                (P-POP-GLOBAL-STEP INS P))
      (POP-LOCN                  (P-POP-LOCN-STEP INS P))
      (POP-CALL                  (P-POP-CALL-STEP INS P))
      (FETCH-TEMP-STK            (P-FETCH-TEMP-STK-STEP INS P))
      (DEPOSIT-TEMP-STK          (P-DEPOSIT-TEMP-STK-STEP INS P))
      (JUMP                      (P-JUMP-STEP INS P))
      (JUMP-CASE                 (P-JUMP-CASE-STEP INS P))
      (PUSHJ                     (P-PUSHJ-STEP INS P))
      (POPJ                      (P-POPJ-STEP INS P))
      (SET-LOCAL                 (P-SET-LOCAL-STEP INS P))
      (SET-GLOBAL                (P-SET-GLOBAL-STEP INS P))
      (TEST-NAT-AND-JUMP         (P-TEST-NAT-AND-JUMP-STEP INS P))
      (TEST-INT-AND-JUMP         (P-TEST-INT-AND-JUMP-STEP INS P))
```

```
      (TEST-BOOL-AND-JUMP       (P-TEST-BOOL-AND-JUMP-STEP INS P))
      (TEST-BITV-AND-JUMP       (P-TEST-BITV-AND-JUMP-STEP INS P))
      (NO-OP                    (P-NO-OP-STEP INS P))
      (ADD-ADDR                 (P-ADD-ADDR-STEP INS P))
      (SUB-ADDR                 (P-SUB-ADDR-STEP INS P))
      (EQ                       (P-EQ-STEP INS P))
      (LT-ADDR                  (P-LT-ADDR-STEP INS P))
      (FETCH                    (P-FETCH-STEP INS P))
      (DEPOSIT                  (P-DEPOSIT-STEP INS P))
      (ADD-INT                  (P-ADD-INT-STEP INS P))
      (ADD-INT-WITH-CARRY       (P-ADD-INT-WITH-CARRY-STEP INS P))
      (ADD1-INT                 (P-ADD1-INT-STEP INS P))
      (SUB-INT                  (P-SUB-INT-STEP INS P))
      (SUB-INT-WITH-CARRY       (P-SUB-INT-WITH-CARRY-STEP INS P))
      (SUB1-INT                 (P-SUB1-INT-STEP INS P))
      (NEG-INT                  (P-NEG-INT-STEP INS P))
      (LT-INT                   (P-LT-INT-STEP INS P))
      (INT-TO-NAT               (P-INT-TO-NAT-STEP INS P))
      (ADD-NAT                  (P-ADD-NAT-STEP INS P))
      (ADD-NAT-WITH-CARRY       (P-ADD-NAT-WITH-CARRY-STEP INS P))
      (ADD1-NAT                 (P-ADD1-NAT-STEP INS P))
      (SUB-NAT                  (P-SUB-NAT-STEP INS P))
      (SUB-NAT-WITH-CARRY       (P-SUB-NAT-WITH-CARRY-STEP INS P))
      (SUB1-NAT                 (P-SUB1-NAT-STEP INS P))
      (LT-NAT                   (P-LT-NAT-STEP INS P))
      (MULT2-NAT                (P-MULT2-NAT-STEP INS P))
      (MULT2-NAT-WITH-CARRY-OUT (P-MULT2-NAT-WITH-CARRY-OUT-STEP INS P))
      (DIV2-NAT                 (P-DIV2-NAT-STEP INS P))
      (OR-BITV                  (P-OR-BITV-STEP INS P))
      (AND-BITV                 (P-AND-BITV-STEP INS P))
      (NOT-BITV                 (P-NOT-BITV-STEP INS P))
      (XOR-BITV                 (P-XOR-BITV-STEP INS P))
      (RSH-BITV                 (P-RSH-BITV-STEP INS P))
      (LSH-BITV                 (P-LSH-BITV-STEP INS P))
      (OR-BOOL                  (P-OR-BOOL-STEP INS P))
      (AND-BOOL                 (P-AND-BOOL-STEP INS P))
      (NOT-BOOL                 (P-NOT-BOOL-STEP INS P))
      (OTHERWISE                (P-HALT P 'RUN)))
```

Definition.

```
(P-INT-TO-NAT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'INT
                     (TOP (P-TEMP-STK P))
                     P)
     (NOT (NEGATIVEP (UNTAG (TOP (P-TEMP-STK P))))))
```

Definition.
```
(P-INT-TO-NAT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'NAT
                    (UNTAG (TOP (P-TEMP-STK P))))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-JUMP-CASE-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'NAT
                     (TOP (P-TEMP-STK P))
                     P)
     (LESSP (UNTAG (TOP (P-TEMP-STK P)))
            (LENGTH (CDR INS))))
```

Definition.
```
(P-JUMP-CASE-STEP INS P)
   =
(P-STATE (PC (GET (UNTAG (TOP (P-TEMP-STK P)))
                  (CDR INS))
             (P-CURRENT-PROGRAM P))
         (P-CTRL-STK P)
         (POP (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-JUMP-IF-TEMP-STK-EMPTY-OKP INS P)
   =
T
```

Definition.
```
(P-JUMP-IF-TEMP-STK-EMPTY-STEP INS P)
   =
(P-STATE (IF (ZEROP (LENGTH (P-TEMP-STK P)))
             (PC (CADR INS) (P-CURRENT-PROGRAM P))
             (ADD1-P-PC P))
         (P-CTRL-STK P)
         (P-TEMP-STK P)
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-JUMP-IF-TEMP-STK-FULL-OKP INS P)
   =
T
```

Definition.
```
(P-JUMP-IF-TEMP-STK-FULL-STEP INS P)
   =
(P-STATE (IF (EQUAL (LENGTH (P-TEMP-STK P))
                    (P-MAX-TEMP-STK-SIZE P))
             (PC (CADR INS) (P-CURRENT-PROGRAM P))
             (ADD1-P-PC P))
         (P-CTRL-STK P)
         (P-TEMP-STK P)
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-JUMP-OKP INS P)
   =
T
```

Definition.
```
(P-JUMP-STEP INS P)
   =
(P-STATE (PC (CADR INS) (P-CURRENT-PROGRAM P))
         (P-CTRL-STK P)
         (P-TEMP-STK P)
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-LOCN-OKP INS P)
   =
(AND (P-OBJECTP-TYPE 'NAT
                     (LOCAL-VAR-VALUE (CADR INS)
                                      (P-CTRL-STK P))
                     P)
     (LESSP (UNTAG (LOCAL-VAR-VALUE (CADR INS)
                                    (P-CTRL-STK P)))
            (LENGTH (BINDINGS (TOP (P-CTRL-STK P)))))
     (LESSP (LENGTH (P-TEMP-STK P))
            (P-MAX-TEMP-STK-SIZE P)))
```

Definition.

```
(P-LOCN-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (CDR (GET (UNTAG (LOCAL-VAR-VALUE (CADR INS)
                                                 (P-CTRL-STK P)))
                         (BINDINGS (TOP (P-CTRL-STK P)))))
               (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.

```
(P-LSH-BITV-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'BITV
                     (TOP (P-TEMP-STK P))
                     P))
```

Definition.

```
(P-LSH-BITV-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'BITV
                    (LSH-BITV (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.

```
(P-LT-ADDR-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'ADDR
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'ADDR
                     (TOP1 (P-TEMP-STK P))
                     P)
     (EQUAL (AREA-NAME (TOP (P-TEMP-STK P)))
            (AREA-NAME (TOP1 (P-TEMP-STK P)))))
```

Definition.
```
(P-LT-ADDR-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (BOOL (LESSP (OFFSET (TOP1 (P-TEMP-STK P)))
                            (OFFSET (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-LT-INT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'INT
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'INT
                     (TOP1 (P-TEMP-STK P))
                     P))
```

Definition.
```
(P-LT-INT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (BOOL (ILESSP (UNTAG (TOP1 (P-TEMP-STK P)))
                             (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-LT-NAT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'NAT
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'NAT
                     (TOP1 (P-TEMP-STK P))
                     P))
```

Definition.
```
(P-LT-NAT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (BOOL (LESSP (UNTAG (TOP1 (P-TEMP-STK P)))
                            (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-MULT2-NAT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'NAT
                     (TOP (P-TEMP-STK P))
                     P)
     (SMALL-NATURALP (TIMES 2 (UNTAG (TOP (P-TEMP-STK P))))
                     (P-WORD-SIZE P)))
```

Definition.
```
(P-MULT2-NAT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'NAT
                    (TIMES 2
                           (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-MULT2-NAT-WITH-CARRY-OUT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'NAT
                     (TOP (P-TEMP-STK P))
                     P)
     (LESSP (LENGTH (P-TEMP-STK P))
            (P-MAX-TEMP-STK-SIZE P)))
```

Definition.

```
(P-MULT2-NAT-WITH-CARRY-OUT-STEP INS P)
   =
(P-STATE
  (ADD1-P-PC P)
  (P-CTRL-STK P)
  (PUSH
   (TAG 'NAT
        (FIX-SMALL-NATURAL (TIMES 2 (UNTAG (TOP (P-TEMP-STK P))))
                           (P-WORD-SIZE P)))
   (PUSH
    (BOOL (NOT (SMALL-NATURALP (TIMES 2 (UNTAG (TOP (P-TEMP-STK P))))
                               (P-WORD-SIZE P))))
    (POP (P-TEMP-STK P))))
  (P-PROG-SEGMENT P)
  (P-DATA-SEGMENT P)
  (P-MAX-CTRL-STK-SIZE P)
  (P-MAX-TEMP-STK-SIZE P)
  (P-WORD-SIZE P)
  'RUN)
```

Definition.

```
(P-NEG-INT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'INT
                     (TOP (P-TEMP-STK P))
                     P)
     (SMALL-INTEGERP (INEGATE (UNTAG (TOP (P-TEMP-STK P))))
                     (P-WORD-SIZE P)))
```

Definition.

```
(P-NEG-INT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'INT
                    (INEGATE (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.

```
(P-NO-OP-OKP INS P)
   =
T
```

Definition.
```
(P-NO-OP-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (P-TEMP-STK P)
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-NOT-BITV-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'BITV
                     (TOP (P-TEMP-STK P))
                     P))
```

Definition.
```
(P-NOT-BITV-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'BITV
                    (NOT-BITV (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-NOT-BOOL-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'BOOL
                     (TOP (P-TEMP-STK P))
                     P))
```

Definition.
```
(P-NOT-BOOL-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'BOOL
                    (NOT-BOOL (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-OBJECTP X P)
   =
(AND (LISTP X)
     (EQUAL (CDDR X) NIL)
     (CASE (TYPE X)
           (NAT       (SMALL-NATURALP (UNTAG X) (P-WORD-SIZE P)))
           (INT       (SMALL-INTEGERP (UNTAG X) (P-WORD-SIZE P)))
           (BITV      (BIT-VECTORP (UNTAG X) (P-WORD-SIZE P)))
           (BOOL      (BOOLEANP (UNTAG X)))
           (ADDR      (ADPP (UNTAG X) (P-DATA-SEGMENT P)))
           (PC        (PCPP (UNTAG X) (P-PROG-SEGMENT P)))
           (SUBR      (DEFINEDP (UNTAG X) (P-PROG-SEGMENT P)))
           (OTHERWISE F)))
```

Definition.
```
(P-OBJECTP-TYPE TYPE X P)
   =
(AND (EQUAL (TYPE X) TYPE)
     (P-OBJECTP X P))
```

Definition.
```
(P-OR-BITV-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'BITV
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'BITV
                     (TOP1 (P-TEMP-STK P))
                     P))
```

Definition.
```
(P-OR-BITV-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'BITV
                    (OR-BITV (UNTAG (TOP1 (P-TEMP-STK P)))
                             (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-OR-BOOL-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'BOOL
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'BOOL
                     (TOP1 (P-TEMP-STK P))
                     P))
```

Definition.
```
(P-OR-BOOL-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'BOOL
                    (OR-BOOL (UNTAG (TOP1 (P-TEMP-STK P)))
                             (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-POP*-OKP INS P)
   =
(NOT (LESSP (LENGTH (P-TEMP-STK P))
            (CADR INS)))
```

Definition.
```
(P-POP*-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (POPN (CADR INS) (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-POP-CALL-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'SUBR
                     (TOP (P-TEMP-STK P))
                     P)
     (P-CALL-OKP (LIST 'CALL
                       (UNTAG (TOP (P-TEMP-STK P))))
                 (P-STATE (P-PC P)
                          (P-CTRL-STK P)
                          (POP (P-TEMP-STK P))
                          (P-PROG-SEGMENT P)
                          (P-DATA-SEGMENT P)
                          (P-MAX-CTRL-STK-SIZE P)
                          (P-MAX-TEMP-STK-SIZE P)
                          (P-WORD-SIZE P)
                          'RUN)))
```

Definition.
```
(P-POP-CALL-STEP INS P)
   =
(P-CALL-STEP (LIST 'CALL
                   (UNTAG (TOP (P-TEMP-STK P))))
             (P-STATE (P-PC P)
                      (P-CTRL-STK P)
                      (POP (P-TEMP-STK P))
                      (P-PROG-SEGMENT P)
                      (P-DATA-SEGMENT P)
                      (P-MAX-CTRL-STK-SIZE P)
                      (P-MAX-TEMP-STK-SIZE P)
                      (P-WORD-SIZE P)
                      'RUN))
```

Definition.
```
(P-POP-GLOBAL-OKP INS P)
   =
(LISTP (P-TEMP-STK P))
```

Definition.
```
(P-POP-GLOBAL-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (POP (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (DEPOSIT (TOP (P-TEMP-STK P))
                  (TAG 'ADDR (CONS (CADR INS) 0))
                  (P-DATA-SEGMENT P))
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-POP-LOCAL-OKP INS P)
   =
(LISTP (P-TEMP-STK P))
```

Definition.
```
(P-POP-LOCAL-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (SET-LOCAL-VAR-VALUE (TOP (P-TEMP-STK P))
                              (CADR INS)
                              (P-CTRL-STK P))
         (POP (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-POP-LOCN-OKP INS P)
   =
(AND (P-OBJECTP-TYPE 'NAT
                     (LOCAL-VAR-VALUE (CADR INS)
                                      (P-CTRL-STK P))
                     P)
     (LESSP (UNTAG (LOCAL-VAR-VALUE (CADR INS)
                                    (P-CTRL-STK P)))
            (LENGTH (BINDINGS (TOP (P-CTRL-STK P)))))
     (LISTP (P-TEMP-STK P)))
```

Definition.
```
(P-POP-LOCN-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (SET-LOCAL-VAR-INDIRECT (TOP (P-TEMP-STK P))
                                 (UNTAG
                                  (LOCAL-VAR-VALUE (CADR INS)
                                                   (P-CTRL-STK P)))
                                 (P-CTRL-STK P))
         (POP (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-POP-OKP INS P)
   =
(LISTP (P-TEMP-STK P))
```

Definition.
```
(P-POP-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (POP (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-POPJ-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'PC
                     (TOP (P-TEMP-STK P))
                     P)
     (EQUAL (AREA-NAME (TOP (P-TEMP-STK P)))
            (AREA-NAME (P-PC P))))
```

Definition.
```
(P-POPJ-STEP INS P)
   =
(P-STATE (TOP (P-TEMP-STK P))
         (P-CTRL-STK P)
         (POP (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-POPN-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'NAT
                     (TOP (P-TEMP-STK P))
                     P)
     (NOT (LESSP (LENGTH (P-TEMP-STK P))
                 (ADD1 (UNTAG (TOP (P-TEMP-STK P)))))))
```

Definition.
```
(P-POPN-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (POPN (UNTAG (TOP (P-TEMP-STK P)))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-PUSH-CONSTANT-OKP INS P)
   =
(LESSP (LENGTH (P-TEMP-STK P))
       (P-MAX-TEMP-STK-SIZE P))
```

Definition.
```
(P-PUSH-CONSTANT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (UNABBREVIATE-CONSTANT (CADR INS) P)
               (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-PUSH-CTRL-STK-FREE-SIZE-OKP INS P)
   =
(LESSP (LENGTH (P-TEMP-STK P))
       (P-MAX-TEMP-STK-SIZE P))
```

Definition.
```
(P-PUSH-CTRL-STK-FREE-SIZE-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'NAT
                    (DIFFERENCE (P-MAX-CTRL-STK-SIZE P)
                                (P-CTRL-STK-SIZE (P-CTRL-STK P))))
               (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-PUSH-GLOBAL-OKP INS P)
   =
(LESSP (LENGTH (P-TEMP-STK P))
       (P-MAX-TEMP-STK-SIZE P))
```

Definition.
```
(P-PUSH-GLOBAL-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (FETCH (TAG 'ADDR (CONS (CADR INS) 0))
                      (P-DATA-SEGMENT P))
               (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-PUSH-LOCAL-OKP INS P)
   =
(LESSP (LENGTH (P-TEMP-STK P))
       (P-MAX-TEMP-STK-SIZE P))
```

Definition.
```
(P-PUSH-LOCAL-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (LOCAL-VAR-VALUE (CADR INS)
                                (P-CTRL-STK P))
               (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-PUSH-TEMP-STK-FREE-SIZE-OKP INS P)
   =
(LESSP (LENGTH (P-TEMP-STK P))
       (P-MAX-TEMP-STK-SIZE P))
```

Definition.
```
(P-PUSH-TEMP-STK-FREE-SIZE-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'NAT
                    (DIFFERENCE (P-MAX-TEMP-STK-SIZE P)
                                (LENGTH (P-TEMP-STK P))))
               (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.

```
(P-PUSH-TEMP-STK-INDEX-OKP INS P)
   =
(AND (LESSP (LENGTH (P-TEMP-STK P))
            (P-MAX-TEMP-STK-SIZE P))
     (LESSP (CADR INS)
            (LENGTH (P-TEMP-STK P))))
```

Definition.

```
(P-PUSH-TEMP-STK-INDEX-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'NAT
                    (SUB1 (DIFFERENCE (LENGTH (P-TEMP-STK P))
                                      (CADR INS))))
               (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.

```
(P-PUSHJ-OKP INS P)
   =
(LESSP (LENGTH (P-TEMP-STK P))
       (P-MAX-TEMP-STK-SIZE P))
```

Definition.

```
(P-PUSHJ-STEP INS P)
   =
(P-STATE (PC (CADR INS) (P-CURRENT-PROGRAM P))
         (P-CTRL-STK P)
         (PUSH (ADD1-P-PC P) (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.

```
(P-RET-OKP INS P)
   =
T
```

Definition.
```
(P-RET-STEP INS P)
   =
(IF (LISTP (POP (P-CTRL-STK P)))
    (P-STATE (RET-PC (TOP (P-CTRL-STK P)))
             (POP (P-CTRL-STK P))
             (P-TEMP-STK P)
             (P-PROG-SEGMENT P)
             (P-DATA-SEGMENT P)
             (P-MAX-CTRL-STK-SIZE P)
             (P-MAX-TEMP-STK-SIZE P)
             (P-WORD-SIZE P)
             'RUN)
    (P-HALT P 'HALT))
```

Definition.
```
(P-RSH-BITV-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'BITV
                     (TOP (P-TEMP-STK P))
                     P))
```

Definition.
```
(P-RSH-BITV-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'BITV
                    (RSH-BITV (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-SET-GLOBAL-OKP INS P)
   =
(LISTP (P-TEMP-STK P))
```

Definition.
```
(P-SET-GLOBAL-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (P-TEMP-STK P)
         (P-PROG-SEGMENT P)
         (DEPOSIT (TOP (P-TEMP-STK P))
                  (TAG 'ADDR (CONS (CADR INS) 0))
                  (P-DATA-SEGMENT P))
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-SET-LOCAL-OKP INS P)
    =
(LISTP (P-TEMP-STK P))
```

Definition.
```
(P-SET-LOCAL-STEP INS P)
    =
(P-STATE (ADD1-P-PC P)
         (SET-LOCAL-VAR-VALUE (TOP (P-TEMP-STK P))
                              (CADR INS)
                              (P-CTRL-STK P))
         (P-TEMP-STK P)
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Shell Definition.
Add the shell **P-STATE** of **9** arguments, with
recognizer function symbol **P-STATEP**, and
accessors **P-PC**, **P-CTRL-STK**, **P-TEMP-STK**, **P-PROG-SEGMENT**,
P-DATA-SEGMENT, **P-MAX-CTRL-STK-SIZE**, **P-MAX-TEMP-STK-SIZE**,
P-WORD-SIZE and **P-PSW**.

Definition.
```
(P-STEP P)
    =
(IF (EQUAL (P-PSW P) 'RUN)
    (P-STEP1 (P-CURRENT-INSTRUCTION P) P)
    P)
```

Definition.
```
(P-STEP1 INS P)
    =
(IF (P-INS-OKP INS P)
    (P-INS-STEP INS P)
    (P-HALT P
            (X-Y-ERROR-MSG 'P (CAR INS))))
```

Definition.
```
(P-SUB-ADDR-OKP INS P)
    =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'NAT
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'ADDR
                     (TOP1 (P-TEMP-STK P))
                     P)
     (NOT (LESSP (OFFSET (TOP1 (P-TEMP-STK P)))
                 (UNTAG (TOP (P-TEMP-STK P))))))
```

Definition.

```
(P-SUB-ADDR-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (SUB-ADDR (TOP1 (P-TEMP-STK P))
                         (UNTAG (TOP (P-TEMP-STK P))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.

```
(P-SUB-INT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'INT
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'INT
                     (TOP1 (P-TEMP-STK P))
                     P)
     (SMALL-INTEGERP (IDIFFERENCE (UNTAG (TOP1 (P-TEMP-STK P)))
                                  (UNTAG (TOP (P-TEMP-STK P))))
                     (P-WORD-SIZE P)))
```

Definition.

```
(P-SUB-INT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'INT
                    (IDIFFERENCE (UNTAG (TOP1 (P-TEMP-STK P)))
                                 (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.

```
(P-SUB-INT-WITH-CARRY-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (LISTP (POP (POP (P-TEMP-STK P))))
     (P-OBJECTP-TYPE 'INT
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'INT
                     (TOP1 (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'BOOL
                     (TOP2 (P-TEMP-STK P))
                     P))
```

Definition.

```
(P-SUB-INT-WITH-CARRY-STEP INS P)
   =
(P-STATE
 (ADD1-P-PC P)
 (P-CTRL-STK P)
 (PUSH
  (TAG 'INT
   (FIX-SMALL-INTEGER
            (IDIFFERENCE (UNTAG (TOP1 (P-TEMP-STK P)))
                         (IPLUS (UNTAG (TOP (P-TEMP-STK P)))
                                (BOOL-TO-NAT
                                 (UNTAG (TOP2 (P-TEMP-STK P))))))
            (P-WORD-SIZE P)))
  (PUSH
   (BOOL
    (NOT
     (SMALL-INTEGERP
            (IDIFFERENCE (UNTAG (TOP1 (P-TEMP-STK P)))
                         (IPLUS (UNTAG (TOP (P-TEMP-STK P)))
                                (BOOL-TO-NAT
                                 (UNTAG (TOP2 (P-TEMP-STK P))))))
            (P-WORD-SIZE P))))
   (POP (POP (POP (P-TEMP-STK P))))))
 (P-PROG-SEGMENT P)
 (P-DATA-SEGMENT P)
 (P-MAX-CTRL-STK-SIZE P)
 (P-MAX-TEMP-STK-SIZE P)
 (P-WORD-SIZE P)
 'RUN)
```

Definition.

```
(P-SUB-NAT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'NAT
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'NAT
                     (TOP1 (P-TEMP-STK P))
                     P)
     (NOT (LESSP (UNTAG (TOP1 (P-TEMP-STK P)))
                 (UNTAG (TOP (P-TEMP-STK P))))))
```

Definition.

```
(P-SUB-NAT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'NAT
                    (DIFFERENCE (UNTAG (TOP1 (P-TEMP-STK P)))
                                (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.

```
(P-SUB-NAT-WITH-CARRY-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (LISTP (POP (POP (P-TEMP-STK P))))
     (P-OBJECTP-TYPE 'NAT
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'NAT
                     (TOP1 (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'BOOL
                     (TOP2 (P-TEMP-STK P))
                     P))
```

Definition.

```
(P-SUB-NAT-WITH-CARRY-STEP INS P)
   =
(P-STATE
 (ADD1-P-PC P)
 (P-CTRL-STK P)
 (PUSH
  (TAG 'NAT
   (IF
     (LESSP (UNTAG (TOP1 (P-TEMP-STK P)))
            (PLUS (UNTAG (TOP (P-TEMP-STK P)))
                  (BOOL-TO-NAT (UNTAG (TOP2 (P-TEMP-STK P))))))
     (DIFFERENCE (EXP 2 (P-WORD-SIZE P))
                 (DIFFERENCE
                  (PLUS
                   (UNTAG (TOP (P-TEMP-STK P)))
                   (BOOL-TO-NAT (UNTAG (TOP2 (P-TEMP-STK P)))))
                  (UNTAG (TOP1 (P-TEMP-STK P)))))
     (DIFFERENCE (UNTAG (TOP1 (P-TEMP-STK P)))
                 (PLUS
                  (UNTAG (TOP (P-TEMP-STK P)))
                  (BOOL-TO-NAT (UNTAG (TOP2 (P-TEMP-STK P))))))))
  (PUSH (BOOL (LESSP (UNTAG (TOP1 (P-TEMP-STK P)))
                     (PLUS
                      (UNTAG (TOP (P-TEMP-STK P)))
                      (BOOL-TO-NAT (UNTAG (TOP2 (P-TEMP-STK P)))))))
        (POP (POP (POP (P-TEMP-STK P))))))
 (P-PROG-SEGMENT P)
 (P-DATA-SEGMENT P)
 (P-MAX-CTRL-STK-SIZE P)
 (P-MAX-TEMP-STK-SIZE P)
 (P-WORD-SIZE P)
 'RUN)
```

Definition.

```
(P-SUB1-INT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'INT
                     (TOP (P-TEMP-STK P))
                     P)
     (SMALL-INTEGERP (IDIFFERENCE (UNTAG (TOP (P-TEMP-STK P)))
                                  1)
                     (P-WORD-SIZE P)))
```

Definition.
```
(P-SUB1-INT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'INT
                    (IDIFFERENCE (UNTAG (TOP (P-TEMP-STK P)))
                                 1))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-SUB1-NAT-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE 'NAT
                     (TOP (P-TEMP-STK P))
                     P)
     (NOT (ZEROP (UNTAG (TOP (P-TEMP-STK P))))))
```

Definition.
```
(P-SUB1-NAT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'NAT
                    (SUB1 (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (P-TEMP-STK P)))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(P-TEST-AND-JUMP-OKP INS TYPE TEST P)
   =
(AND (LISTP (P-TEMP-STK P))
     (P-OBJECTP-TYPE TYPE
                     (TOP (P-TEMP-STK P))
                     P))
```

Definition.

```
(P-TEST-AND-JUMP-STEP TEST LAB P)
   =
(IF TEST
    (P-STATE (PC LAB (P-CURRENT-PROGRAM P))
             (P-CTRL-STK P)
             (POP (P-TEMP-STK P))
             (P-PROG-SEGMENT P)
             (P-DATA-SEGMENT P)
             (P-MAX-CTRL-STK-SIZE P)
             (P-MAX-TEMP-STK-SIZE P)
             (P-WORD-SIZE P)
             'RUN)
    (P-STATE (ADD1-P-PC P)
             (P-CTRL-STK P)
             (POP (P-TEMP-STK P))
             (P-PROG-SEGMENT P)
             (P-DATA-SEGMENT P)
             (P-MAX-CTRL-STK-SIZE P)
             (P-MAX-TEMP-STK-SIZE P)
             (P-WORD-SIZE P)
             'RUN))
```

Definition.

```
(P-TEST-BITV-AND-JUMP-OKP INS P)
   =
(P-TEST-AND-JUMP-OKP INS 'BITV
                     (P-TEST-BITVP (CADR INS)
                                   (UNTAG (TOP (P-TEMP-STK P))))
                     P)
```

Definition.

```
(P-TEST-BITV-AND-JUMP-STEP INS P)
   =
(P-TEST-AND-JUMP-STEP (P-TEST-BITVP (CADR INS)
                                    (UNTAG (TOP (P-TEMP-STK P))))
                      (CADDR INS)
                      P)
```

Definition.

```
(P-TEST-BITVP FLG X)
   =
(IF (EQUAL FLG 'ALL-ZERO)
    (ALL-ZERO-BITVP X)
    (NOT (ALL-ZERO-BITVP X)))
```

Definition.

```
(P-TEST-BOOL-AND-JUMP-OKP INS P)
   =
(P-TEST-AND-JUMP-OKP INS 'BOOL
                     (P-TEST-BOOLP (CADR INS)
                                   (UNTAG (TOP (P-TEMP-STK P))))
                     P)
```

Definition.

```
(P-TEST-BOOL-AND-JUMP-STEP INS P)
   =
(P-TEST-AND-JUMP-STEP (P-TEST-BOOLP (CADR INS)
                                    (UNTAG (TOP (P-TEMP-STK P))))
                      (CADDR INS)
                      P)
```

Definition.

```
(P-TEST-BOOLP FLG X)
   =
(IF (EQUAL FLG 'T)
    (EQUAL X 'T)
    (EQUAL X 'F))
```

Definition.

```
(P-TEST-INT-AND-JUMP-OKP INS P)
   =
(P-TEST-AND-JUMP-OKP INS 'INT
                     (P-TEST-INTP (CADR INS)
                                  (UNTAG (TOP (P-TEMP-STK P))))
                     P)
```

Definition.

```
(P-TEST-INT-AND-JUMP-STEP INS P)
   =
(P-TEST-AND-JUMP-STEP (P-TEST-INTP (CADR INS)
                                   (UNTAG (TOP (P-TEMP-STK P))))
                      (CADDR INS)
                      P)
```

Definition.

```
(P-TEST-INTP FLG X)
   =
(CASE FLG
      (ZERO      (EQUAL X 0))
      (NOT-ZERO  (NOT (EQUAL X 0)))
      (NEG       (NEGATIVEP X))
      (NOT-NEG   (NOT (NEGATIVEP X)))
      (POS (AND  (NUMBERP X) (NOT (EQUAL X 0))))
      (OTHERWISE (OR (EQUAL X 0) (NEGATIVEP X))))
```

Definition.

```
(P-TEST-NAT-AND-JUMP-OKP INS P)
   =
(P-TEST-AND-JUMP-OKP INS 'NAT
                     (P-TEST-NATP (CADR INS)
                                  (UNTAG (TOP (P-TEMP-STK P))))
                     P)
```

Definition.

```
(P-TEST-NAT-AND-JUMP-STEP INS P)
   =
(P-TEST-AND-JUMP-STEP (P-TEST-NATP (CADR INS)
                                   (UNTAG (TOP (P-TEMP-STK P))))
                      (CADDR INS)
                      P)
```

Definition.
```
(P-TEST-NATP FLG X)
   =
(IF (EQUAL FLG 'ZERO)
    (EQUAL X 0)
    (NOT (EQUAL X 0)))
```

Definition.
```
(P-XOR-BITV-OKP INS P)
   =
(AND (LISTP (P-TEMP-STK P))
     (LISTP (POP (P-TEMP-STK P)))
     (P-OBJECTP-TYPE 'BITV
                     (TOP (P-TEMP-STK P))
                     P)
     (P-OBJECTP-TYPE 'BITV
                     (TOP1 (P-TEMP-STK P))
                     P))
```

Definition.
```
(P-XOR-BITV-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (TAG 'BITV
                    (XOR-BITV (UNTAG (TOP1 (P-TEMP-STK P)))
                              (UNTAG (TOP (P-TEMP-STK P)))))
               (POP (POP (P-TEMP-STK P))))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN)
```

Definition.
```
(PAIR-FORMAL-VARS-WITH-ACTUALS FORMAL-VARS TEMP-STK)
   =
(PAIRLIST FORMAL-VARS
          (REVERSE (FIRST-N (LENGTH FORMAL-VARS)
                            TEMP-STK)))
```

Definition.
```
(PAIR-TEMPS-WITH-INITIAL-VALUES TEMP-VAR-DCLS)
   =
(IF (NLISTP TEMP-VAR-DCLS)
    NIL
    (CONS (CONS (CAAR TEMP-VAR-DCLS)
                (CADAR TEMP-VAR-DCLS))
          (PAIR-TEMPS-WITH-INITIAL-VALUES (CDR TEMP-VAR-DCLS))))
```

Definition.
```
(PC LAB PROGRAM)
   =
(TAG 'PC
     (CONS (NAME PROGRAM)
           (FIND-LABEL LAB
                       (PROGRAM-BODY PROGRAM))))
```

Definition.
```
(PCPP X SEGMENT)
   =
(AND (LISTP X)
     (NUMBERP (ADP-OFFSET X))
     (DEFINEDP (ADP-NAME X) SEGMENT)
     (LESSP (ADP-OFFSET X)
            (LENGTH (PROGRAM-BODY (DEFINITION (ADP-NAME X) SEGMENT)))))
```

Definition.
```
(POP STK)
   =
(CDR STK)
```

Definition.
```
(POPN N X)
   =
(IF (ZEROP N)
    X
    (POPN (SUB1 N) (CDR X)))
```

Definition.
```
(PROGRAM-BODY D)
   =
(CDDDR D)
```

Definition.
```
(PROPER-LABELED-P-INSTRUCTIONSP LST NAME P)
   =
(IF (NLISTP LST)
    (EQUAL LST NIL)
    (AND (LEGAL-LABELP (CAR LST))
         (PROPER-P-INSTRUCTIONP (UNLABEL (CAR LST))
                                NAME P)
         (PROPER-LABELED-P-INSTRUCTIONSP (CDR LST)
                                         NAME P)))
```

Definition.
```
(PROPER-P-ADD-ADDR-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-ADD-INT-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-ADD-INT-WITH-CARRY-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-ADD-NAT-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-ADD-NAT-WITH-CARRY-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-ADD1-INT-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-ADD1-NAT-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-ALISTP ALIST P)
   =
(IF (NLISTP ALIST)
    (EQUAL ALIST NIL)
    (AND (LISTP (CAR ALIST))
         (LITATOM (CAAR ALIST))
         (P-OBJECTP (CDAR ALIST) P)
         (PROPER-P-ALISTP (CDR ALIST) P)))
```

Definition.
```
(PROPER-P-AND-BITV-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-AND-BOOL-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-AREA AREA P)
   =
(AND (LITATOM (CAR AREA))
     (LISTP (CDR AREA))
     (ALL-P-OBJECTPS (CDR AREA) P))
```

Definition.
```
(PROPER-P-CALL-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 2)
     (DEFINEDP (CADR INS)
               (P-PROG-SEGMENT P)))
```

Definition.
```
(PROPER-P-CTRL-STKP CTRL-STK NAME P)
   =
(IF (NLISTP CTRL-STK)
    (EQUAL CTRL-STK NIL)
    (AND (PROPER-P-FRAMEP (TOP CTRL-STK)
                          NAME P)
         (PROPER-P-CTRL-STKP (POP CTRL-STK)
                             (AREA-NAME (RET-PC (TOP CTRL-STK)))
                             P)))
```

Definition.

```
(PROPER-P-DATA-SEGMENTP DATA-SEGMENT P)
   =
(IF (NLISTP DATA-SEGMENT)
    (EQUAL DATA-SEGMENT NIL)
    (AND (PROPER-P-AREA (CAR DATA-SEGMENT) P)
         (NOT (DEFINEDP (CAAR DATA-SEGMENT)
                        (CDR DATA-SEGMENT)))
         (PROPER-P-DATA-SEGMENTP (CDR DATA-SEGMENT)
                                 P)))
```

Definition.

```
(PROPER-P-DEPOSIT-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPER-P-DEPOSIT-TEMP-STK-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPER-P-DIV2-NAT-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPER-P-EQ-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPER-P-FETCH-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPER-P-FETCH-TEMP-STK-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPER-P-FRAMEP FRAME NAME P)
   =
(AND (LISTP FRAME)
     (LISTP (CDR FRAME))
     (EQUAL (CDDR FRAME) NIL)
     (PROPER-P-ALISTP (BINDINGS FRAME) P)
     (EQUAL (STRIP-CARS (BINDINGS FRAME))
            (LOCAL-VARS (DEFINITION NAME (P-PROG-SEGMENT P))))
     (P-OBJECTP-TYPE 'PC
                     (RET-PC FRAME)
                     P))
```

Definition.

```
(PROPER-P-INSTRUCTIONP INS NAME P)
   =
(AND
 (PROPERP INS)
 (CASE
  (CAR INS)
  (CALL             (PROPER-P-CALL-INSTRUCTIONP INS NAME P))
  (RET              (PROPER-P-RET-INSTRUCTIONP INS NAME P))
  (LOCN             (PROPER-P-LOCN-INSTRUCTIONP INS NAME P))
  (PUSH-CONSTANT    (PROPER-P-PUSH-CONSTANT-INSTRUCTIONP INS NAME P))
  (PUSH-LOCAL       (PROPER-P-PUSH-LOCAL-INSTRUCTIONP INS NAME P))
  (PUSH-GLOBAL      (PROPER-P-PUSH-GLOBAL-INSTRUCTIONP INS NAME P))

  (PUSH-CTRL-STK-FREE-SIZE
                    (PROPER-P-PUSH-CTRL-STK-FREE-SIZE-INSTRUCTIONP
                              INS NAME P))
  (PUSH-TEMP-STK-FREE-SIZE
                    (PROPER-P-PUSH-TEMP-STK-FREE-SIZE-INSTRUCTIONP
                              INS NAME P))
  (PUSH-TEMP-STK-INDEX
                    (PROPER-P-PUSH-TEMP-STK-INDEX-INSTRUCTIONP
                              INS NAME P))
  (JUMP-IF-TEMP-STK-FULL
                    (PROPER-P-JUMP-IF-TEMP-STK-FULL-INSTRUCTIONP
                              INS NAME P))
  (JUMP-IF-TEMP-STK-EMPTY
                    (PROPER-P-JUMP-IF-TEMP-STK-EMPTY-INSTRUCTIONP
                              INS NAME P))
  (POP              (PROPER-P-POP-INSTRUCTIONP INS NAME P))
  (POP*             (PROPER-P-POP*-INSTRUCTIONP INS NAME P))
  (POPN             (PROPER-P-POPN-INSTRUCTIONP INS NAME P))
  (POP-LOCAL        (PROPER-P-POP-LOCAL-INSTRUCTIONP INS NAME P))
  (POP-GLOBAL       (PROPER-P-POP-GLOBAL-INSTRUCTIONP INS NAME P))
  (POP-LOCN         (PROPER-P-POP-LOCN-INSTRUCTIONP INS NAME P))
  (POP-CALL         (PROPER-P-POP-CALL-INSTRUCTIONP INS NAME P))
  (FETCH-TEMP-STK   (PROPER-P-FETCH-TEMP-STK-INSTRUCTIONP INS NAME P))
  (DEPOSIT-TEMP-STK (PROPER-P-DEPOSIT-TEMP-STK-INSTRUCTIONP INS NAME P))
  (JUMP             (PROPER-P-JUMP-INSTRUCTIONP INS NAME P))
  (JUMP-CASE        (PROPER-P-JUMP-CASE-INSTRUCTIONP INS NAME P))
  (PUSHJ            (PROPER-P-PUSHJ-INSTRUCTIONP INS NAME P))
  (POPJ             (PROPER-P-POPJ-INSTRUCTIONP INS NAME P))
  (SET-LOCAL        (PROPER-P-SET-LOCAL-INSTRUCTIONP INS NAME P))
  (SET-GLOBAL       (PROPER-P-SET-GLOBAL-INSTRUCTIONP INS NAME P))
  (TEST-NAT-AND-JUMP
                    (PROPER-P-TEST-NAT-AND-JUMP-INSTRUCTIONP
                              INS NAME P))
  (TEST-INT-AND-JUMP
                    (PROPER-P-TEST-INT-AND-JUMP-INSTRUCTIONP
                              INS NAME P))
  (TEST-BOOL-AND-JUMP
                    (PROPER-P-TEST-BOOL-AND-JUMP-INSTRUCTIONP
                              INS NAME P))
  (TEST-BITV-AND-JUMP
                    (PROPER-P-TEST-BITV-AND-JUMP-INSTRUCTIONP
                              INS NAME P))
  (NO-OP            (PROPER-P-NO-OP-INSTRUCTIONP INS NAME P))
```

```
  (ADD-ADDR         (PROPER-P-ADD-ADDR-INSTRUCTIONP INS NAME P))
  (SUB-ADDR         (PROPER-P-SUB-ADDR-INSTRUCTIONP INS NAME P))
  (EQ               (PROPER-P-EQ-INSTRUCTIONP INS NAME P))
  (LT-ADDR          (PROPER-P-LT-ADDR-INSTRUCTIONP INS NAME P))
  (FETCH            (PROPER-P-FETCH-INSTRUCTIONP INS NAME P))
  (DEPOSIT          (PROPER-P-DEPOSIT-INSTRUCTIONP INS NAME P))
  (ADD-INT          (PROPER-P-ADD-INT-INSTRUCTIONP INS NAME P))
  (ADD-INT-WITH-CARRY
                    (PROPER-P-ADD-INT-WITH-CARRY-INSTRUCTIONP
                              INS NAME P))
  (ADD1-INT         (PROPER-P-ADD1-INT-INSTRUCTIONP INS NAME P))
  (SUB-INT          (PROPER-P-SUB-INT-INSTRUCTIONP INS NAME P))
  (SUB-INT-WITH-CARRY
                    (PROPER-P-SUB-INT-WITH-CARRY-INSTRUCTIONP
                              INS NAME P))
  (SUB1-INT         (PROPER-P-SUB1-INT-INSTRUCTIONP INS NAME P))
  (NEG-INT          (PROPER-P-NEG-INT-INSTRUCTIONP INS NAME P))
  (LT-INT           (PROPER-P-LT-INT-INSTRUCTIONP INS NAME P))
  (INT-TO-NAT       (PROPER-P-INT-TO-NAT-INSTRUCTIONP INS NAME P))
  (ADD-NAT          (PROPER-P-ADD-NAT-INSTRUCTIONP INS NAME P))
  (ADD-NAT-WITH-CARRY
                    (PROPER-P-ADD-NAT-WITH-CARRY-INSTRUCTIONP
                              INS NAME P))
  (ADD1-NAT         (PROPER-P-ADD1-NAT-INSTRUCTIONP INS NAME P))
  (SUB-NAT          (PROPER-P-SUB-NAT-INSTRUCTIONP INS NAME P))
  (SUB-NAT-WITH-CARRY
                    (PROPER-P-SUB-NAT-WITH-CARRY-INSTRUCTIONP
                              INS NAME P))
  (SUB1-NAT         (PROPER-P-SUB1-NAT-INSTRUCTIONP INS NAME P))
  (LT-NAT           (PROPER-P-LT-NAT-INSTRUCTIONP INS NAME P))
  (MULT2-NAT        (PROPER-P-MULT2-NAT-INSTRUCTIONP INS NAME P))
  (MULT2-NAT-WITH-CARRY-OUT
                    (PROPER-P-MULT2-NAT-WITH-CARRY-OUT-INSTRUCTIONP
                              INS NAME P))
  (DIV2-NAT         (PROPER-P-DIV2-NAT-INSTRUCTIONP INS NAME P))
  (OR-BITV          (PROPER-P-OR-BITV-INSTRUCTIONP INS NAME P))
  (AND-BITV         (PROPER-P-AND-BITV-INSTRUCTIONP INS NAME P))
  (NOT-BITV         (PROPER-P-NOT-BITV-INSTRUCTIONP INS NAME P))
  (XOR-BITV         (PROPER-P-XOR-BITV-INSTRUCTIONP INS NAME P))
  (RSH-BITV         (PROPER-P-RSH-BITV-INSTRUCTIONP INS NAME P))
  (LSH-BITV         (PROPER-P-LSH-BITV-INSTRUCTIONP INS NAME P))
  (OR-BOOL          (PROPER-P-OR-BOOL-INSTRUCTIONP INS NAME P))
  (AND-BOOL         (PROPER-P-AND-BOOL-INSTRUCTIONP INS NAME P))
  (NOT-BOOL         (PROPER-P-NOT-BOOL-INSTRUCTIONP INS NAME P))
  (OTHERWISE        F)))
```

Definition.

```
(PROPER-P-INT-TO-NAT-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPER-P-JUMP-CASE-INSTRUCTIONP INS NAME P)
   =
(AND (LISTP (CDR INS))
     (ALL-FIND-LABELP (CDR INS)
                      (PROGRAM-BODY (DEFINITION NAME
                                                (P-PROG-SEGMENT P)))))
```

Definition.

```
(PROPER-P-JUMP-IF-TEMP-STK-EMPTY-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 2)
     (FIND-LABELP (CADR INS)
                  (PROGRAM-BODY (DEFINITION NAME
                                            (P-PROG-SEGMENT P)))))
```

Definition.

```
(PROPER-P-JUMP-IF-TEMP-STK-FULL-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 2)
     (FIND-LABELP (CADR INS)
                  (PROGRAM-BODY (DEFINITION NAME
                                            (P-PROG-SEGMENT P)))))
```

Definition.

```
(PROPER-P-JUMP-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 2)
     (FIND-LABELP (CADR INS)
                  (PROGRAM-BODY (DEFINITION NAME
                                            (P-PROG-SEGMENT P)))))
```

Definition.

```
(PROPER-P-LOCN-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 2)
     (MEMBER (CADR INS)
             (LOCAL-VARS (DEFINITION NAME
                                     (P-PROG-SEGMENT P)))))
```

Definition.

```
(PROPER-P-LSH-BITV-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPER-P-LT-ADDR-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPER-P-LT-INT-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPER-P-LT-NAT-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPER-P-MULT2-NAT-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPER-P-MULT2-NAT-WITH-CARRY-OUT-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-NEG-INT-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-NO-OP-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-NOT-BITV-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-NOT-BOOL-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-OR-BITV-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-OR-BOOL-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-POP*-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 2)
     (SMALL-NATURALP (CADR INS)
                     (P-WORD-SIZE P)))
```

Definition.
```
(PROPER-P-POP-CALL-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-POP-GLOBAL-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 2)
     (DEFINEDP (CADR INS)
               (P-DATA-SEGMENT P)))
```

Definition.
```
(PROPER-P-POP-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-POP-LOCAL-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 2)
     (MEMBER (CADR INS)
             (LOCAL-VARS (DEFINITION NAME
                                     (P-PROG-SEGMENT P)))))
```

Definition.
```
(PROPER-P-POP-LOCN-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 2)
     (MEMBER (CADR INS)
             (LOCAL-VARS (DEFINITION NAME
                                     (P-PROG-SEGMENT P)))))
```

Definition.
```
(PROPER-P-POPJ-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-POPN-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-PROG-SEGMENTP SEGMENT P)
   =
(IF (NLISTP SEGMENT)
    (EQUAL SEGMENT NIL)
    (AND (PROPER-P-PROGRAMP (CAR SEGMENT) P)
         (PROPER-P-PROG-SEGMENTP (CDR SEGMENT)
                                 P)))
```

Definition.
```
(PROPER-P-PROGRAM-BODYP LST NAME P)
   =
(AND (LISTP LST)
     (PROPER-LABELED-P-INSTRUCTIONSP LST NAME P)
     (FALL-OFF-PROOFP LST))
```

Definition.
```
(PROPER-P-PROGRAMP PROG P)
   =
(AND (LITATOM (NAME PROG))
     (ALL-LITATOMS (FORMAL-VARS PROG))
     (PROPER-P-TEMP-VAR-DCLSP (TEMP-VAR-DCLS PROG)
                              P)
     (PROPER-P-PROGRAM-BODYP (PROGRAM-BODY PROG)
                             (NAME PROG)
                             P))
```

Definition.
```
(PROPER-P-PUSH-CONSTANT-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 2)
     (OR (P-OBJECTP (CADR INS) P)
         (EQUAL (CADR INS) 'PC)
         (FIND-LABELP (CADR INS)
                      (PROGRAM-BODY (DEFINITION NAME
                                                (P-PROG-SEGMENT P))))))
```

Definition.
```
(PROPER-P-PUSH-CTRL-STK-FREE-SIZE-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-PUSH-GLOBAL-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 2)
     (DEFINEDP (CADR INS)
               (P-DATA-SEGMENT P)))
```

Definition.
```
(PROPER-P-PUSH-LOCAL-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 2)
     (MEMBER (CADR INS)
             (LOCAL-VARS (DEFINITION NAME
                                     (P-PROG-SEGMENT P)))))
```

Definition.
```
(PROPER-P-PUSH-TEMP-STK-FREE-SIZE-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-PUSH-TEMP-STK-INDEX-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 2)
     (SMALL-NATURALP (CADR INS)
                     (P-WORD-SIZE P)))
```

Definition.
```
(PROPER-P-PUSHJ-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 2)
     (FIND-LABELP (CADR INS)
                  (PROGRAM-BODY (DEFINITION NAME
                                            (P-PROG-SEGMENT P)))))
```

Definition.
```
(PROPER-P-RET-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-RSH-BITV-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-SET-GLOBAL-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 2)
     (DEFINEDP (CADR INS)
               (P-DATA-SEGMENT P)))
```

Definition.
```
(PROPER-P-SET-LOCAL-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 2)
     (MEMBER (CADR INS)
             (LOCAL-VARS (DEFINITION NAME
                                     (P-PROG-SEGMENT P)))))
```

Definition.

```
(PROPER-P-STATEP P)
   =
(AND (P-STATEP P)
     (P-OBJECTP-TYPE 'PC (P-PC P) P)
     (LISTP (P-CTRL-STK P))
     (PROPER-P-FRAMEP (TOP (P-CTRL-STK P))
                      (AREA-NAME (P-PC P))
                      P)
     (PROPER-P-CTRL-STKP (POP (P-CTRL-STK P))
                         (AREA-NAME (RET-PC (TOP (P-CTRL-STK P))))
                         P)
     (NOT (LESSP (P-MAX-CTRL-STK-SIZE P)
                 (P-CTRL-STK-SIZE (P-CTRL-STK P))))
     (PROPER-P-TEMP-STKP (P-TEMP-STK P) P)
     (NOT (LESSP (P-MAX-TEMP-STK-SIZE P)
                 (LENGTH (P-TEMP-STK P))))
     (PROPER-P-PROG-SEGMENTP (P-PROG-SEGMENT P)
                             P)
     (PROPER-P-DATA-SEGMENTP (P-DATA-SEGMENT P)
                             P)
     (NUMBERP (P-MAX-CTRL-STK-SIZE P))
     (NUMBERP (P-MAX-TEMP-STK-SIZE P))
     (NUMBERP (P-WORD-SIZE P))
     (LESSP (P-MAX-CTRL-STK-SIZE P)
            (EXP 2 (P-WORD-SIZE P)))
     (LESSP (P-MAX-TEMP-STK-SIZE P)
            (EXP 2 (P-WORD-SIZE P)))
     (LESSP 0 (P-WORD-SIZE P)))
```

Definition.

```
(PROPER-P-SUB-ADDR-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPER-P-SUB-INT-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPER-P-SUB-INT-WITH-CARRY-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPER-P-SUB-NAT-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPER-P-SUB-NAT-WITH-CARRY-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPER-P-SUB1-INT-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-SUB1-NAT-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.
```
(PROPER-P-TEMP-STKP TEMP-STK P)
   =
(IF (NLISTP TEMP-STK)
    (EQUAL TEMP-STK NIL)
    (AND (P-OBJECTP (TOP TEMP-STK) P)
         (PROPER-P-TEMP-STKP (POP TEMP-STK)
                             P)))
```

Definition.
```
(PROPER-P-TEMP-VAR-DCLSP TEMP-VAR-DCLS P)
   =
(IF (NLISTP TEMP-VAR-DCLS)
    T
    (AND (LITATOM (CAAR TEMP-VAR-DCLS))
         (P-OBJECTP (CADAR TEMP-VAR-DCLS) P)
         (PROPER-P-TEMP-VAR-DCLSP (CDR TEMP-VAR-DCLS)
                                  P)))
```

Definition.
```
(PROPER-P-TEST-BITV-AND-JUMP-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 3)
     (FIND-LABELP (CADDR INS)
                  (PROGRAM-BODY (DEFINITION NAME
                                            (P-PROG-SEGMENT P)))))
```

Definition.
```
(PROPER-P-TEST-BOOL-AND-JUMP-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 3)
     (FIND-LABELP (CADDR INS)
                  (PROGRAM-BODY (DEFINITION NAME
                                            (P-PROG-SEGMENT P)))))
```

Definition.
```
(PROPER-P-TEST-INT-AND-JUMP-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 3)
     (FIND-LABELP (CADDR INS)
                  (PROGRAM-BODY (DEFINITION NAME
                                            (P-PROG-SEGMENT P)))))
```

Definition.
```
(PROPER-P-TEST-NAT-AND-JUMP-INSTRUCTIONP INS NAME P)
   =
(AND (EQUAL (LENGTH INS) 3)
     (FIND-LABELP (CADDR INS)
                  (PROGRAM-BODY (DEFINITION NAME
                                            (P-PROG-SEGMENT P)))))
```

Definition.
```
(PROPER-P-XOR-BITV-INSTRUCTIONP INS NAME P)
   =
(EQUAL (LENGTH INS) 1)
```

Definition.

```
(PROPERP X)
   =
(IF (NLISTP X)
    (EQUAL X NIL)
    (PROPERP (CDR X)))
```

Definition.

```
(PUSH X STK)
   =
(CONS X STK)
```

Definition.

```
(PUT VAL N LST)
   =
(IF (ZEROP N)
    (IF (LISTP LST)
        (CONS VAL (CDR LST))
        (LIST VAL))
    (CONS (CAR LST)
          (PUT VAL (SUB1 N) (CDR LST))))
```

Definition.

```
(PUT-ASSOC VAL NAME ALIST)
   =
(COND ((NLISTP ALIST) ALIST)
      ((EQUAL NAME (CAAR ALIST))
       (CONS (CONS NAME VAL) (CDR ALIST)))
      (T (CONS (CAR ALIST)
               (PUT-ASSOC VAL NAME (CDR ALIST)))))
```

Definition.

```
(PUT-VALUE VAL NAME ALIST)
   =
(PUT-ASSOC VAL NAME ALIST)
```

Definition.

```
(PUT-VALUE-INDIRECT VAL N LST)
   =
(IF (LISTP LST)
    (IF (ZEROP N)
        (CONS (CONS (CAAR LST) VAL) (CDR LST))
        (CONS (CAR LST)
              (PUT-VALUE-INDIRECT VAL
                                  (SUB1 N)
                                  (CDR LST))))
    LST)
```

Definition.

```
(RET-PC FRAME)
   =
(CADR FRAME)
```

Definition.

```
(REVERSE X)
   =
(IF (NLISTP X)
    NIL
    (APPEND (REVERSE (CDR X))
            (LIST (CAR X))))
```

Definition.
```
(RGET N LST)
   =
(GET (SUB1 (DIFFERENCE (LENGTH LST) N))
     LST)
```

Definition.
```
(RPUT VAL N LST)
   =
(PUT VAL
     (SUB1 (DIFFERENCE (LENGTH LST) N))
     LST)
```

Definition.
```
(RSH-BITV A)
   =
(CONS 0 (ALL-BUT-LAST A))
```

Definition.
```
(SET-LOCAL-VAR-INDIRECT VAL INDEX CTRL-STK)
   =
(PUSH (P-FRAME (PUT-VALUE-INDIRECT VAL INDEX
                                   (BINDINGS (TOP CTRL-STK)))
               (RET-PC (TOP CTRL-STK)))
      (POP CTRL-STK))
```

Definition.
```
(SET-LOCAL-VAR-VALUE VAL VAR CTRL-STK)
   =
(PUSH (P-FRAME (PUT-VALUE VAL VAR
                          (BINDINGS (TOP CTRL-STK)))
               (RET-PC (TOP CTRL-STK)))
      (POP CTRL-STK))
```

Definition.
```
(SMALL-INTEGERP I WORD-SIZE)
   =
(AND (INTEGERP I)
     (NOT (ILESSP I
                  (MINUS (EXP 2 (SUB1 WORD-SIZE)))))
     (ILESSP I (EXP 2 (SUB1 WORD-SIZE))))
```

Definition.
```
(SMALL-NATURALP I WORD-SIZE)
   =
(AND (NUMBERP I)
     (LESSP I (EXP 2 WORD-SIZE)))
```

Definition.
```
(SUB-ADDR ADDR N)
   =
(TAG (TYPE ADDR)
     (SUB-ADP (UNTAG ADDR) N))
```

Definition.
```
(SUB-ADP ADP N)
   =
(CONS (ADP-NAME ADP)
      (DIFFERENCE (ADP-OFFSET ADP) N))
```

Definition.
```
(TAG TYPE OBJ)
   =
(LIST TYPE OBJ)
```

Definition.
```
(TEMP-VAR-DCLS D)
   =
(CADDR D)
```

Definition.
```
(TOP STK)
   =
(CAR STK)
```

Definition.
```
(TOP1 STK)
   =
(TOP (POP STK))
```

Definition.
```
(TOP2 STK)
   =
(TOP (POP (POP STK)))
```

Definition.
```
(TYPE CONST)
   =
(CAR CONST)
```

Definition.
```
(UNABBREVIATE-CONSTANT C P)
   =
(COND ((EQUAL C 'PC) (ADD1-P-PC P))
      ((NLISTP C)
       (PC C (P-CURRENT-PROGRAM P)))
      (T C))
```

Definition.
```
(UNLABEL X)
   =
(IF (LABELLEDP X) (CADDDR X) X)
```

Definition.
```
(UNTAG CONST)
   =
(CADR CONST)
```

Definition.
```
(VALUE NAME ALIST)
   =
(CDR (DEFINITION NAME ALIST))
```

Definition.
```
(X-Y-ERROR-MSG X Y)
   =
(PACK (APPEND (UNPACK 'ILLEGAL-)
              (APPEND (UNPACK Y)
                      (CDR (UNPACK 'G-INSTRUCTION)))))
```

Definition.

```
(XOR-BIT BIT1 BIT2)
   =
(COND ((EQUAL BIT1 0)
       (IF (EQUAL BIT2 0) 0 1))
      ((EQUAL BIT2 0) 1)
      (T 0))
```

Definition.

```
(XOR-BITV A B)
   =
(IF (NLISTP A)
    NIL
    (CONS (XOR-BIT (CAR A) (CAR B))
          (XOR-BITV (CDR A) (CDR B))))
```

8. The Formal Definition of FM8502

In this chapter we present the formal definition of the FM8502. As before, the definitions are presented in alphabetical order and are fully indexed and cross-indexed.

The exponentiation function, **EXP**, is displayed both here and in the preceding formal definition of Piton, simply so that both chapters are self-contained. **EXP** is the only function defined by both of these chapters.

8.1. A Guide to the Formal Definition of FM8502

The state of the FM8502 is formally represented by the shell objects of type **M-STATE**

Shell Definition.
Add the shell **M-STATE** of **6** arguments, with
recognizer function symbol **M-STATEP**, and
accessors **M-REGS**, **M-C-FLG**, **M-V-FLG**, **M-N-FLG**, **M-Z-FLG**
and **M-MEM**.

The six components of an **M-STATE** are, respectively, the list of eight registers, the four flags, and the memory.

The formal definition of FM8502 is

Definition.
```
(FM8502 STATE N)
    =
(FM8502->M (SOFT (M-REGS STATE)
                 (M-MEM STATE)
                 (M-C-FLG STATE)
                 (M-V-FLG STATE)
                 (M-Z-FLG STATE)
                 (M-N-FLG STATE)
                 (FM8502-ORACLE N))).
```

FM8502 is given an initial m-state, **STATE**, and a natural number **N** indicating how many machine instructions to execute. **FM8502** first ‘‘destructures’’ the state into its six components. It uses **FM8502-ORACLE** to convert the natural number **N** into a list of **N** zeros—this is used to specify how many instructions are to be executed by the ‘‘software’’ machine **SOFT**; the elements of the list are unimportant and do not affect **SOFT**’s final answer.[16] **FM8502** then uses **SOFT** to determine the values of the six components of the final state. Finally, **FM8502->M** is used to package up the six values into a single m-state again.

Our **SOFT** is so similar to Hunt’s, which is explained in detail in [11], that we do not further elaborate. We do include the entire tree of definitions in this report. We did so for three reasons. First, we thus enter it into the historical record. Second, by glancing through it the reader can see that **FM8502** is a radically more primitive machine than the Piton machine. (Just look at all those bit vector operations!) Third, by simply measuring the thickness of this Chapter the reader can confirm that **FM8502** is (in some technical sense) a simple machine.

---

[16]That **SOFT** takes a list instead of a number as its last argument is a reflection of the fact that **SOFT** is proved correct with respect to a gate level implementation. In that proof the elements of the oracle are used to specify the lengths of the various wait states the machine enters.

Readers uninterested in pursuing the formal definition of the FM8502 at this time should skip to page 169.

8.2. Alphabetical Listing of the FM8502 Definitions

Definition.
```
(A-VALUE-FOR-ALU-AFTER-OPRD-A-PRE-DECREMENT REG-FILE REAL-MEM)
   =
(COND
 ((B-DIRECT-REG-A (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
  (V-NTH (BV-OPRD-A (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
         (REG-FILE-AFTER-PC-INCREMENT REG-FILE)))
 ((B-INDIRECT-REG-A-DEC (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
  (V-NTH
   (V-NAT-DEC
    (V-NTH (BV-OPRD-A (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
           (REG-FILE-AFTER-PC-INCREMENT REG-FILE)))
   REAL-MEM))
 (T (V-NTH
     (V-NTH (BV-OPRD-A (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
            (REG-FILE-AFTER-PC-INCREMENT REG-FILE))
     REAL-MEM)))
```

Definition.
```
(B-AND X Y)
   =
(AND X Y)
```

Definition.
```
(B-C-SET I-REG)
   =
(BITN I-REG 14)
```

Definition.
```
(B-DIRECT-REG-A I-REG)
   =
(B-NOR (BITN I-REG 4) (BITN I-REG 5))
```

Definition.
```
(B-DIRECT-REG-B I-REG)
   =
(B-NOR (BITN I-REG 9) (BITN I-REG 10))
```

Definition.
```
(B-EQUV X Y)
   =
(COND (X (IF Y T F)) (Y F) (T T))
```

Definition.
```
(B-IF C A B)
   =
(B-NAND (B-NAND C A)
        (B-NAND (B-NOT C) B))
```

Definition.
```
(B-INDIRECT-REG-A-DEC I-REG)
   =
(B-AND (B-NOT (BITN I-REG 4))
       (BITN I-REG 5))
```

Definition.
```
(B-INDIRECT-REG-A-INC I-REG)
   =
(B-AND (BITN I-REG 4) (BITN I-REG 5))
```

Definition.
```
(B-INDIRECT-REG-B-DEC I-REG)
   =
(B-AND (B-NOT (BITN I-REG 9))
       (BITN I-REG 10))
```

Definition.
```
(B-INDIRECT-REG-B-INC I-REG)
   =
(B-AND (BITN I-REG 9) (BITN I-REG 10))
```

Definition.
```
(B-MOVE-OP I-REG)
   =
(BITN I-REG 16)
```

Definition.
```
(B-N-SET I-REG)
   =
(BITN I-REG 12)
```

Definition.
```
(B-NAND X Y)
   =
(IF X (IF Y F T) T)
```

Definition.
```
(B-NOR X Y)
   =
(COND (X F) (Y F) (T T))
```

Definition.
```
(B-NOT X)
   =
(IF X F T)
```

Definition.
```
(B-OR X Y)
   =
(OR X Y)
```

Definition.
```
(B-STORE-ALU-RESULT-WITH-IFS C-FLAG V-FLAG Z-FLAG N-FLAG I-REG)
   =
(OR (NOT (B-MOVE-OP I-REG))
    (BITN (BV-OP-CODE I-REG) 4)
    (EQUAL (BV-OP-CODE I-REG)
           (NAT-TO-BV (IF C-FLAG 1 0) 4))
    (EQUAL (BV-OP-CODE I-REG)
           (NAT-TO-BV (IF V-FLAG 3 2) 4))
    (EQUAL (BV-OP-CODE I-REG)
           (NAT-TO-BV (IF Z-FLAG 5 4) 4))
    (EQUAL (BV-OP-CODE I-REG)
           (NAT-TO-BV (IF N-FLAG 7 6) 4)))
```

Definition.
```
(B-V-SET I-REG)
```
=
```
(BITN I-REG 13)
```

Definition.
```
(B-VALUE-FOR-ALU-AFTER-OPRD-B-PRE-DECREMENT REG-FILE REAL-MEM)
```
=
```
(COND
 ((B-DIRECT-REG-B (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
  (V-NTH (BV-OPRD-B (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
         (REG-FILE-AFTER-OPRD-A-PRE-DECREMENT REG-FILE REAL-MEM)))
 ((B-INDIRECT-REG-B-DEC (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
  (V-NTH
   (V-NAT-DEC
    (V-NTH (BV-OPRD-B (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
           (REG-FILE-AFTER-OPRD-A-PRE-DECREMENT REG-FILE REAL-MEM)))
   REAL-MEM))
 (T (V-NTH
     (V-NTH (BV-OPRD-B (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
            (REG-FILE-AFTER-OPRD-A-PRE-DECREMENT REG-FILE REAL-MEM))
     REAL-MEM)))
```

Definition.
```
(B-XOR X Y)
```
=
```
(COND (X (IF Y F T)) (Y T) (T F))
```

Definition.
```
(B-Z-SET I-REG)
```
=
```
(BITN I-REG 11)
```

Definition.
```
(BITN X N)
```
=
```
(COND ((ZEROP N) F)
      ((EQUAL N 1) (BIT X))
      (T (BITN (VEC X) (SUB1 N))))
```

Shell Definition.
Add the shell **BITV** of **2** arguments, with
base function symbol **BTM**,
recognizer function symbol **BITVP**,
accessors **BIT** and **VEC**,
type restrictions **(ONE-OF TRUEP FALSEP)** and **(ONE-OF BITVP)**, and
default function symbols **FALSE** and **BTM**.

Definition.
```
(BTMP A)
```
=
```
(IF (BITVP A) (EQUAL A (BTM)) T)
```

Definition.
```
(BV-ADDER C A B)
   =
(IF (BITVP A)
    (IF (EQUAL A (BTM))
        (BITV C (BTM))
        (BITV (B-XOR C (B-XOR (BIT A) (BIT B)))
              (BV-ADDER (B-OR (B-AND (BIT A) (BIT B))
                              (B-OR (B-AND (BIT A) C)
                                    (B-AND (BIT B) C)))
                        (VEC A)
                        (VEC B))))
    (BITV C (BTM)))
```

Definition.
```
(BV-ADDER-CARRY-OUT C A B)
   =
(BITN (BV-ADDER C A B)
      (ADD1 (SIZE A)))
```

Definition.
```
(BV-ADDER-OUTPUT C A B)
   =
(TRUNC (BV-ADDER C A B) (SIZE A))
```

Definition.
```
(BV-ADDER-OVERFLOWP C A B)
   =
(B-AND (B-EQUV (BITN A (SIZE A))
               (BITN B (SIZE B)))
       (B-XOR (BITN A (SIZE A))
              (BITN (BV-ADDER-OUTPUT C A B)
                    (SIZE A))))
```

Definition.
```
(BV-ALU-CV A B C OP-CODE)
   =
(BV-CV-IF (BITN OP-CODE 4)
          (BV-CV-IF (BITN OP-CODE 3)
                    (BV-CV-IF (BITN OP-CODE 2)
                              (BV-CV-IF (BITN OP-CODE 1)
                                        (BV-CV A F F)
                                        (BV-CV (BV-NOT A) F F))
                              (BV-CV-IF (BITN OP-CODE 1)
                                        (BV-CV (BV-AND A B) F F)
                                        (BV-CV (BV-OR A B) F F)))
                    (BV-CV-IF (BITN OP-CODE 2)
                              (BV-CV-IF (BITN OP-CODE 1)
                                        (BV-CV (BV-XOR A B) F F)
                                        (BV-CV (BV-LSR A) (BITN A 1) F))
                              (BV-CV-IF (BITN OP-CODE 1)
                                        (BV-CV (BV-ASR A) (BITN A 1) F)
                                        (BV-CV (BV-ROR A C)
                                               (IF (ZEROP (SIZE A))
                                                   C
                                                   (BITN A 1))
                                               F))))
          (BV-CV-IF (BITN OP-CODE 3)
                    (BV-CV-IF (BITN OP-CODE 2)
```

```
                                        (BV-CV-IF
                                         (BITN OP-CODE 1)
                                         (BV-CV (BV-SUBTRACTER-OUTPUT F A B)
                                                (BV-SUBTRACTER-CARRY-OUT F A B)
                                                (BV-SUBTRACTER-OVERFLOWP F A B))
                                         (BV-CV (BV-SUBTRACTER-OUTPUT C A B)
                                                (BV-SUBTRACTER-CARRY-OUT C A B)
                                                (BV-SUBTRACTER-OVERFLOWP C A B)))
                                        (BV-CV-IF
                                         (BITN OP-CODE 1)
                                         (BV-CV (BV-SUBTRACTER-OUTPUT
                                                 T (NAT-TO-BV 0 (SIZE A)) A)
                                                (BV-SUBTRACTER-CARRY-OUT
                                                 T (NAT-TO-BV 0 (SIZE A)) A)
                                                (BV-SUBTRACTER-OVERFLOWP
                                                 T (NAT-TO-BV 0 (SIZE A)) A))
                                         (BV-CV (BV-SUBTRACTER-OUTPUT
                                                 F A (NAT-TO-BV 0 (SIZE A)))
                                                (BV-SUBTRACTER-CARRY-OUT
                                                 F A (NAT-TO-BV 0 (SIZE A)))
                                                (BV-SUBTRACTER-OVERFLOWP
                                                 F A (NAT-TO-BV 0 (SIZE A))))))
                              (BV-CV-IF (BITN OP-CODE 2)
                                        (BV-CV-IF
                                         (BITN OP-CODE 1)
                                         (BV-CV (BV-ADDER-OUTPUT F A B)
                                                (BV-ADDER-CARRY-OUT F A B)
                                                (BV-ADDER-OVERFLOWP F A B))
                                         (BV-CV (BV-ADDER-OUTPUT C A B)
                                                (BV-ADDER-CARRY-OUT C A B)
                                                (BV-ADDER-OVERFLOWP C A B)))
                                        (BV-CV-IF
                                         (BITN OP-CODE 1)
                                         (BV-CV (BV-ADDER-OUTPUT
                                                 T A (NAT-TO-BV 0 (SIZE A)))
                                                (BV-ADDER-CARRY-OUT
                                                 T A (NAT-TO-BV 0 (SIZE A)))
                                                (BV-ADDER-OVERFLOWP
                                                 T A (NAT-TO-BV 0 (SIZE A))))
                                         (BV-CV A F F)))))
```

Definition.

```
(BV-ALU-CV-RESULTS REG-FILE REAL-MEM C-FLAG)
   =
(BV-ALU-CV
    (A-VALUE-FOR-ALU-AFTER-OPRD-A-PRE-DECREMENT REG-FILE REAL-MEM)
    (B-VALUE-FOR-ALU-AFTER-OPRD-B-PRE-DECREMENT REG-FILE REAL-MEM)
    C-FLAG
    (BV-ALU-OP-CODE (CURRENT-INSTRUCTION REG-FILE REAL-MEM)))
```

Definition.

```
(BV-ALU-OP-CODE I-REG)
   =
(BV-IF (B-MOVE-OP I-REG)
       (NAT-TO-BV 0 4)
       (BV-OP-CODE I-REG))
```

Definition.

```
(BV-AND A B)
   =
(IF (BTMP A)
    (BTM)
    (BITV (B-AND (BIT A) (BIT B))
          (BV-AND (VEC A) (VEC B))))
```

Definition.

```
(BV-ASR A)
   =
(V-ASR A)
```

Shell Definition.
Add the shell **BV-CV** of **3** arguments, with
recognizer function symbol **BV-CVP**,
accessors **BV**, **C** and **V**,
type restrictions **(ONE-OF BITVP)**, **(ONE-OF TRUEP FALSEP)** and
(ONE-OF TRUEP FALSEP), and
default function symbols **BTM**, **FALSE** and **FALSE**.

Definition.

```
(BV-CV-IF C A B)
   =
(BV-CV (BV-IF C (BV A) (BV B))
       (B-IF C (C A) (C B))
       (B-IF C (V A) (V B)))
```

Definition.

```
(BV-IF C A B)
   =
(IF (BTMP A)
    (BTM)
    (BITV (B-IF C (BIT A) (BIT B))
          (BV-IF C (VEC A) (VEC B))))
```

Definition.

```
(BV-LSR A)
   =
(V-LSR A)
```

Definition.

```
(BV-NOT A)
   =
(IF (BTMP A)
    (BTM)
    (BITV (B-NOT (BIT A))
          (BV-NOT (VEC A))))
```

Definition.
```
(BV-OP-CODE I-REG)
   =
(BITV (BITN I-REG 17)
      (BITV (BITN I-REG 18)
            (BITV (BITN I-REG 19)
                  (BITV (BITN I-REG 20) (BTM)))))
```

Definition.
```
(BV-OPRD-A I-REG)
   =
(BITV (BITN I-REG 1)
      (BITV (BITN I-REG 2)
            (BITV (BITN I-REG 3) (BTM))))
```

Definition.
```
(BV-OPRD-B I-REG)
   =
(BITV (BITN I-REG 6)
      (BITV (BITN I-REG 7)
            (BITV (BITN I-REG 8) (BTM))))
```

Definition.
```
(BV-OR A B)
   =
(IF (BTMP A)
    (BTM)
    (BITV (B-OR (BIT A) (BIT B))
          (BV-OR (VEC A) (VEC B))))
```

Definition.
```
(BV-ROR A C)
   =
(V-ROR A C)
```

Definition.
```
(BV-SUBTRACTER-CARRY-OUT C A B)
   =
(B-NOT (BV-ADDER-CARRY-OUT (B-NOT C)
                           (BV-NOT A)
                           B))
```

Definition.
```
(BV-SUBTRACTER-OUTPUT C A B)
   =
(BV-ADDER-OUTPUT (B-NOT C)
                 (BV-NOT A)
                 B)
```

Definition.
```
(BV-SUBTRACTER-OVERFLOWP C A B)
   =
(BV-ADDER-OVERFLOWP (B-NOT C)
                    (BV-NOT A)
                    B)
```

Definition.

```
(BV-TO-NAT X)
   =
(IF (BTMP X)
    0
    (PLUS (IF (BIT X) 1 0)
          (TIMES 2 (BV-TO-NAT (VEC X)))))
```

Definition.

```
(BV-TO-TC X)
   =
(IF (BITN X (SIZE X))
    (MINUS (BV-TO-NAT (INCR T (COMPL X))))
    (BV-TO-NAT X))
```

Definition.

```
(BV-XOR A B)
   =
(IF (BTMP A)
    (BTM)
    (BITV (B-XOR (BIT A) (BIT B))
          (BV-XOR (VEC A) (VEC B))))
```

Definition.

```
(COMPL X)
   =
(IF (BITVP X)
    (IF (EQUAL X (BTM))
        (BTM)
        (BITV (NOT (BIT X)) (COMPL (VEC X))))
    (BTM))
```

Definition.

```
(CURRENT-INSTRUCTION REG-FILE REAL-MEM)
   =
(V-NTH (NTH 0 REG-FILE) REAL-MEM)
```

Definition.

```
(EXP I J)
   =
(IF (ZEROP J)
    1
    (TIMES I (EXP I (SUB1 J))))
```

Definition.

```
(FM8502 STATE N)
   =
(FM8502->M (SOFT (M-REGS STATE)
                 (M-MEM STATE)
                 (M-C-FLG STATE)
                 (M-V-FLG STATE)
                 (M-Z-FLG STATE)
                 (M-N-FLG STATE)
                 (FM8502-ORACLE N)))
```

Definition .
```
(FM8502->M TUPLE)
   =
(M-STATE (NTH 0 TUPLE)
         (NTH 2 TUPLE)
         (NTH 3 TUPLE)
         (NTH 5 TUPLE)
         (NTH 4 TUPLE)
         (NTH 1 TUPLE))
```

Definition .
```
(FM8502-ORACLE N)
   =
(IF (ZEROP N)
    NIL
    (CONS 0 (FM8502-ORACLE (SUB1 N))))
```

Definition .
```
(INCR C X)
   =
(IF (BITVP X)
    (IF (EQUAL X (BTM))
        (BTM)
        (BITV (XOR C (BIT X))
              (INCR (AND C (BIT X)) (VEC X))))
    (BTM))
```

Shell Definition .
Add the shell **M-STATE** of **6** arguments, with recognizer function symbol **M-STATEP**, and accessors **M-REGS**, **M-C-FLG**, **M-V-FLG**, **M-N-FLG**, **M-Z-FLG** and **M-MEM**.

Definition .
```
(NAT-TO-BV N SIZE)
   =
(IF (ZEROP SIZE)
    (BTM)
    (BITV (IF (ZEROP (REMAINDER N 2)) F T)
          (NAT-TO-BV (QUOTIENT N 2)
                     (SUB1 SIZE))))
```

Definition .
```
(NTH N LST)
   =
(IF (ZEROP N)
    (CAR LST)
    (NTH (SUB1 N) (CDR LST)))
```

Definition.

```
(REAL-MEM-AFTER-ALU-WRITE REG-FILE REAL-MEM
                          C-FLAG V-FLAG Z-FLAG N-FLAG)
   =
(UPDATE-V-NTH
 (AND (B-STORE-ALU-RESULT-WITH-IFS
                                C-FLAG V-FLAG Z-FLAG N-FLAG
                                (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
      (NOT (B-DIRECT-REG-B (CURRENT-INSTRUCTION REG-FILE REAL-MEM))))
 (V-NTH (BV-OPRD-B (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
        (REG-FILE-AFTER-OPRD-B-PRE-DECREMENT REG-FILE REAL-MEM))
 REAL-MEM
 (BV (BV-ALU-CV-RESULTS REG-FILE REAL-MEM C-FLAG)))
```

Definition.

```
(REG-FILE-AFTER-ALU-WRITE REG-FILE REAL-MEM
                          C-FLAG V-FLAG Z-FLAG N-FLAG)
   =
(UPDATE-V-NTH
 (AND (B-STORE-ALU-RESULT-WITH-IFS
                          C-FLAG V-FLAG Z-FLAG N-FLAG
                          (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
      (B-DIRECT-REG-B (CURRENT-INSTRUCTION REG-FILE REAL-MEM)))
 (BV-OPRD-B (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
 (REG-FILE-AFTER-OPRD-B-PRE-DECREMENT REG-FILE REAL-MEM)
 (BV (BV-ALU-CV-RESULTS REG-FILE REAL-MEM C-FLAG)))
```

Definition.

```
(REG-FILE-AFTER-OPRD-A-POST-INCREMENT REG-FILE REAL-MEM
                                      C-FLAG V-FLAG Z-FLAG N-FLAG)
   =
(UPDATE-V-NTH
 (B-INDIRECT-REG-A-INC (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
 (BV-OPRD-A (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
 (REG-FILE-AFTER-ALU-WRITE REG-FILE REAL-MEM
                           C-FLAG V-FLAG Z-FLAG N-FLAG)
 (V-NAT-INC (V-NTH (BV-OPRD-A (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
                   (REG-FILE-AFTER-ALU-WRITE
                                         REG-FILE REAL-MEM
                                         C-FLAG V-FLAG Z-FLAG N-FLAG))))
```

Definition.

```
(REG-FILE-AFTER-OPRD-A-PRE-DECREMENT REG-FILE REAL-MEM)
   =
(UPDATE-V-NTH
 (B-INDIRECT-REG-A-DEC (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
 (BV-OPRD-A (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
 (REG-FILE-AFTER-PC-INCREMENT REG-FILE)
 (V-NAT-DEC (V-NTH (BV-OPRD-A (CURRENT-INSTRUCTION REG-FILE
                                                   REAL-MEM))
                   (REG-FILE-AFTER-PC-INCREMENT REG-FILE))))
```

Definition.

```
(REG-FILE-AFTER-OPRD-B-POST-INCREMENT REG-FILE REAL-MEM
                                      C-FLAG V-FLAG Z-FLAG N-FLAG)
   =
(UPDATE-V-NTH
 (B-INDIRECT-REG-B-INC (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
 (BV-OPRD-B (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
 (REG-FILE-AFTER-OPRD-A-POST-INCREMENT REG-FILE REAL-MEM
                                       C-FLAG V-FLAG Z-FLAG N-FLAG)
 (V-NAT-INC (V-NTH (BV-OPRD-B (CURRENT-INSTRUCTION REG-FILE
                                                   REAL-MEM))
                   (REG-FILE-AFTER-OPRD-A-POST-INCREMENT REG-FILE
                                                         REAL-MEM
                                                         C-FLAG
                                                         V-FLAG
                                                         Z-FLAG
                                                         N-FLAG))))
```

Definition.

```
(REG-FILE-AFTER-OPRD-B-PRE-DECREMENT REG-FILE REAL-MEM)
   =
(UPDATE-V-NTH
 (B-INDIRECT-REG-B-DEC (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
 (BV-OPRD-B (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
 (REG-FILE-AFTER-OPRD-A-PRE-DECREMENT REG-FILE REAL-MEM)
 (V-NAT-DEC (V-NTH (BV-OPRD-B (CURRENT-INSTRUCTION REG-FILE
                                                   REAL-MEM))
                   (REG-FILE-AFTER-OPRD-A-PRE-DECREMENT REG-FILE
                                                        REAL-MEM))))
```

Definition.

```
(REG-FILE-AFTER-PC-INCREMENT REG-FILE)
   =
(UPDATE-NTH T
            0
            REG-FILE
            (V-NAT-INC (NTH 0 REG-FILE)))
```

Definition.

```
(SIZE A)
   =
(IF (BITVP A)
    (IF (EQUAL A (BTM))
        0
        (ADD1 (SIZE (VEC A))))
    0)
```

Definition.

```
(SOFT REG-FILE REAL-MEM C-FLAG V-FLAG Z-FLAG N-FLAG LST)
   =
(IF (NLISTP LST)
    (LIST REG-FILE REAL-MEM C-FLAG V-FLAG Z-FLAG N-FLAG)
    (SOFT (REG-FILE-AFTER-OPRD-B-POST-INCREMENT
                                   REG-FILE REAL-MEM
                                   C-FLAG V-FLAG Z-FLAG N-FLAG)
          (REAL-MEM-AFTER-ALU-WRITE REG-FILE REAL-MEM
                                   C-FLAG V-FLAG Z-FLAG N-FLAG)
          (UPDATE-V (B-C-SET (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
                    C-FLAG
                    (C (BV-ALU-CV-RESULTS REG-FILE REAL-MEM C-FLAG)))
          (UPDATE-V (B-V-SET (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
                    V-FLAG
                    (V (BV-ALU-CV-RESULTS REG-FILE REAL-MEM C-FLAG)))
          (UPDATE-V (B-Z-SET (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
                    Z-FLAG
                    (ZEROP (BV-TO-NAT (BV (BV-ALU-CV-RESULTS REG-FILE
                                                             REAL-MEM
                                                             C-FLAG)))))
          (UPDATE-V (B-N-SET (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
                    N-FLAG
                    (NEGATIVEP
                     (BV-TO-TC (BV (BV-ALU-CV-RESULTS REG-FILE
                                                      REAL-MEM
                                                      C-FLAG)))))
          (CDR LST)))
```

Definition.

```
(TRUNC A N)
   =
(IF (ZEROP N)
    (BTM)
    (BITV (BIT A)
          (TRUNC (VEC A) (SUB1 N))))
```

Definition.

```
(UPDATE-NTH C N LST VALUE)
   =
(IF (AND (TRUEP C) (LISTP LST))
    (IF (ZEROP N)
        (CONS VALUE (CDR LST))
        (CONS (CAR LST)
              (UPDATE-NTH C
                          (SUB1 N)
                          (CDR LST)
                          VALUE)))
    LST)
```

Definition.

```
(UPDATE-V C CELL VALUE)
   =
(IF (TRUEP C) VALUE CELL)
```

Definition.
```
(UPDATE-V-NTH C V-N LST VALUE)
   =
(UPDATE-NTH C
            (BV-TO-NAT V-N)
            LST VALUE)
```

Definition.
```
(V-APPEND A B)
   =
(IF (BTMP A)
    B
    (BITV (BIT A) (V-APPEND (VEC A) B)))
```

Definition.
```
(V-ASR A)
   =
(IF (BTMP A)
    (BTM)
    (V-APPEND (VEC A)
              (BITV (BITN A (SIZE A)) (BTM))))
```

Definition.
```
(V-LSR A)
   =
(IF (BTMP A)
    (BTM)
    (V-APPEND (VEC A) (BITV F (BTM))))
```

Definition.
```
(V-NAT-DEC A)
   =
(IF (ZEROP (BV-TO-NAT A))
    (NAT-TO-BV (SUB1 (EXP 2 (SIZE A)))
               (SIZE A))
    (NAT-TO-BV (SUB1 (BV-TO-NAT A))
               (SIZE A)))
```

Definition.
```
(V-NAT-INC A)
   =
(NAT-TO-BV (ADD1 (BV-TO-NAT A))
           (SIZE A))
```

Definition.
```
(V-NTH V-N LST)
   =
(NTH (BV-TO-NAT V-N) LST)
```

Definition.
```
(V-ROR A C)
   =
(IF (BTMP A)
    (BTM)
    (V-APPEND (VEC A) (BITV C (BTM))))
```

Definition.
```
(XOR X Y)
   =
(COND (X (IF Y F T)) (Y T) (T F))
```

9. The Formal Implementation

In this chapter we formally present the implementation of Piton on the FM8502. As before, we divide the chapter into two parts: a guide to the formal listing and then the listing itself.

To make this chapter self-contained we have included here the definitions of approximately three dozen minor utility functions that are also displayed in the formal presentation of Piton or in the formal presentation of FM8502. The functions in the first category are: **ADP-NAME**, **ADP-OFFSET**, **AREA-NAME**, **BINDINGS**, **DEFINITION**, **EXP**, **FIND-LABEL**, **FORMAL-VARS**, **LABELLEDP**, **LENGTH**, **LOCAL-VARS**, **NAME**, **P-CTRL-STK-SIZE**, **P-FRAME-SIZE**, **P-STATE**, **PC**, **POP**, **PROGRAM-BODY**, **RET-PC**, **REVERSE**, **SUB-ADDR**, **SUB-ADP**, **TAG**, **TEMP-VAR-DCLS**, **TOP**, **TYPE**, **UNLABEL** and **UNTAG**. The functions in the second category are: **BITV**, **BTMP**, **COMPL**, **EXP**, **INCR**, **M-STATE**, **NAT-TO-BV**, **V-APPEND**, and **XOR**. These redundant definitions consume about 180 lines of text, less than 3 pages.

9.1. A Guide to the Formal Implementation

The FM8502 implementation of Piton is expressed in the function **LOAD**,

Definition.

```
(LOAD P)
    =
(I->M (R->I (P->R P))).
```

Observe that **LOAD** is the composition of three functions. The innermost, called **P->R**, is the implementation of the **r**esource representation phase; it maps a p-state into what we call an *r-state*. In the next phase we compile the Piton source code into symbolic i-code. This is implemented by the function **R->I**, which maps an r-state into what we call an *i-state*. The final phase, implemented by **I->M**, does link-assembling and maps i-states into m-states.

We discuss each phase in turn.

9.1.1. The Formal Definition of Resource Representation

R-states are represented by the new shell class

Shell Definition.
Add the shell **R-STATE** of **15** arguments, with
recognizer function symbol **R-STATEP**, and
accessors **R-PC**, **R-CFP**, **R-CSP**, **R-TSP**, **R-X**, **R-Y**,
R-C-FLG, **R-V-FLG**, **R-N-FLG**, **R-Z-FLG**,
R-PROG-SEGMENT, **R-USR-DATA-SEGMENT**, **R-SYS-DATA-SEGMENT**,
R-WORD-SIZE and **R-PSW**.

The r-state corresponding to a given p-state has exactly the same program counter, program segment, data segment (here called ‘‘user data segment’’), word size and psw. However, r-states also have five additional ‘‘registers’’ (the **cfp**, **csp**, **tsp**, **x** and **y** registers), four ‘‘flag registers,’’ and a ‘‘system data segment.’’ These components of an r-state specify how the corresponding resources of the underlying FM8502 machine will be used to implement the resources of the Piton machine.

The r-state corresponding to a given p-state is produced by the function **P->R**, which implements the resource representation phase of **LOAD**.

Definition.
```
(P->R P)
   =
(R-STATE (P-PC P)
         (P->R_CFP (P-CTRL-STK P)
                   (P-MAX-CTRL-STK-SIZE P))
         (P->R_CSP (P-CTRL-STK P)
                   (P-MAX-CTRL-STK-SIZE P))
         (P->R_TSP (P-TEMP-STK P)
                   (P-MAX-TEMP-STK-SIZE P))
         '(NAT 0)
         '(NAT 0)
         '(BOOL F)
         '(BOOL F)
         '(BOOL F)
         '(BOOL F)
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P->R_SYS-DATA-SEGMENT (P-CTRL-STK P)
                                (P-MAX-CTRL-STK-SIZE P)
                                (P-TEMP-STK P)
                                (P-MAX-TEMP-STK-SIZE P))
         (P-WORD-SIZE P)
         (P-PSW P))
```

Observe that the program counter, program segment, data segment, word size and psw of the given p-state are copied as is into the r-state. The three stack registers are initialized by the three functions **P->R_CFP**, **P->R_CSP**, and **P->R_TSP**. The two temporary registers are initialized to the natural number 0 and the four flags are all initially **F**. The system data segment of the r-state is determined by **P->R_SYS-DATA-SEGMENT**.

We will look in detail at the handling of the temporary stack, since it is the simpler of the two stacks. The initial stack pointer, the **tsp** register, is computed by

Definition.
```
(P->R_TSP STK MAX)
   =
(TAG 'SYS-ADDR
     (CONS 'TSTK
           (DIFFERENCE MAX (LENGTH STK)))).
```

Note that this function yields a system data address into the **TSTK** area. The arithmetic accounts for the fact that the stack is loaded into the high end of the **TSTK** area.

The system data segment itself is constructed by

Definition .
```
(P->R_SYS-DATA-SEGMENT CTRL-STK MAX-CTRL-STK-SIZE
                       TEMP-STK MAX-TEMP-STK-SIZE)
   =
(LIST (P->R_CTRL-STK CTRL-STK MAX-CTRL-STK-SIZE)
      (P->R_TEMP-STK TEMP-STK MAX-TEMP-STK-SIZE)
      (LIST 'FULL-CTRL-STK-ADDR
            (TAG 'SYS-ADDR '(CSTK . 0)))
      (LIST 'FULL-TEMP-STK-ADDR
            (TAG 'SYS-ADDR '(TSTK . 0)))
      (LIST 'EMPTY-TEMP-STK-ADDR
            (TAG 'SYS-ADDR
                 (CONS 'TSTK MAX-TEMP-STK-SIZE)))),
```

which describes a segment with five areas. The second one is the temporary stack area, constructed by

Definition .
```
(P->R_TEMP-STK TEMP-STK MAX-TEMP-STK-SIZE)
   =
(CONS 'TSTK
      (APPEND (NAT-0S (DIFFERENCE MAX-TEMP-STK-SIZE
                                  (LENGTH TEMP-STK)))
              (APPEND TEMP-STK
                      (LIST (TAG 'NAT 0))))).
```

Observe that the temporary stack area is named **TSTK** and is associated with an array of length **MAX-TEMP-STK-SIZE**+1. The array is initialized with the temporary stack from the p-state, padded at the high end of the area by a single natural number 0 and padded on the low end by as many 0's as necessary to make the array the correct length.

The construction of the control stack area is much more subtle but easily followed from the informal discussion.

9.1.2. The Formal Definition of the Compiler

The compilation phase translates Piton to i-code. The state transformation associated with this is implemented by **R->I**, which converts an r-state into an i-state, where

Shell Definition .
Add the shell **I-STATE** of **15** arguments, with
recognizer function symbol **I-STATEP**, and
accessors **I-PC**, **I-CFP**, **I-CSP**,**I-TSP**, **I-X**, **I-Y**,
I-C-FLG, **I-V-FLG**, **I-N-FLG**,**I-Z-FLG**,
I-PROG-SEGMENT, **I-USR-DATA-SEGMENT**,**I-SYS-DATA-SEGMENT**,
I-WORD-SIZE and **I-PSW**.

The i-state corresponding to a given r-state is identical in all components except the program counter, the program segment, and the psw. The code in the i-state program segment is produced by compiling the Piton code in the r-state program segment. The program counter of the i-state is arranged to point to the beginning of the i-code block for the current instruction in the r-state. The psw of the i-state is either **RUN** or an error—**HALT** is coerced to **RUN**.

The definition of **R->I** is

Definition.
```
(R->I R)
    =
(I-STATE (R->I_PC (R-PC R) (R-PROG-SEGMENT R))
         (R-CFP R)
         (R-CSP R)
         (R-TSP R)
         (R-X R)
         (R-Y R)
         (R-C-FLG R)
         (R-V-FLG R)
         (R-N-FLG R)
         (R-Z-FLG R)
         (ICOMPILE (R-PROG-SEGMENT R))
         (R-USR-DATA-SEGMENT R)
         (R-SYS-DATA-SEGMENT R)
         (R-WORD-SIZE R)
         (R->I_PSW (R-PSW R))).
```

The compiler is **ICOMPILE**.

Definition.
```
(ICOMPILE PROGRAMS)
    =
(IF (NLISTP PROGRAMS)
    NIL
    (CONS (ICOMPILE-PROGRAM (CAR PROGRAMS))
          (ICOMPILE (CDR PROGRAMS))))
```

This function maps over the list of programs and compiles each with **ICOMPILE-PROGRAM**.

Definition.
```
(ICOMPILE-PROGRAM PROGRAM)
    =
(CONS (NAME PROGRAM)
      (APPEND (GENERATE-PRELUDE PROGRAM)
              (APPEND (ICOMPILE-PROGRAM-BODY (PROGRAM-BODY PROGRAM)
                                             0 PROGRAM)
                      (GENERATE-POSTLUDE PROGRAM))))
```

ICOMPILE-PROGRAM conses the program name onto the result of compiling the body of the program and sandwiching it between the prelude and the postlude. The cons pair returned becomes the definition of a program area in the program segment of the i-state.

The prelude is generated with **GENERATE-PRELUDE**,

Definition.
```
(GENERATE-PRELUDE PROGRAM)
    =
(APPEND (LIST (DL (CONS (NAME PROGRAM) '(PRELUDE))
                  '(PRELUDE)
                  '(CPUSH_CFP))
              '(MOVE_CFP_CSP))
        (APPEND (GENERATE-PRELUDE1 (REVERSE (TEMP-VAR-DCLS PROGRAM)))
                (GENERATE-PRELUDE2 (FORMAL-VARS PROGRAM)))).
```

Note that the block of code generated by **GENERATE-PRELUDE** begins with a def-label form defining a label of the form **(name PRELUDE)**. The first instruction is the **CPUSH_CFP** that pushes the old cfp word of the new frame. The next instruction saves **csp** in **cfp**. Then we lay down the code to initialize the temporary variables (generated by **GENERATE-PRELUDE1**) and the formals (**GENERATE-PRELUDE2**).

Below is **GENERATE-PRELUDE1**. It generates a list of **CPUSH_*** instructions, each of which is followed by the initial value specified in the temporary variable declaration of the program being compiled.

Definition.

```
(GENERATE-PRELUDE1 TEMP-VAR-DCLS)
    =
(IF (NLISTP TEMP-VAR-DCLS)
    NIL
    (CONS '(CPUSH_*)
          (CONS (CADAR TEMP-VAR-DCLS)
                (GENERATE-PRELUDE1 (CDR TEMP-VAR-DCLS)))))
```

GENERATE-PRELUDE2 generates a list of **CPUSH_<TSP>+** instructions as long as the formals of the program. These instructions move the actual parameters from the temporary stack to the control stack.

Definition.

```
(GENERATE-PRELUDE2 FORMAL-VARS)
    =
(IF (NLISTP FORMAL-VARS)
    NIL
    (CONS '(CPUSH_<TSP>+)
          (GENERATE-PRELUDE2 (CDR FORMAL-VARS))))
```

The generation of the postlude is similar in spirit to the examples given here.

The body of each program is compiled with

Definition.

```
(ICOMPILE-PROGRAM-BODY LST PCN PROGRAM)
    =
(IF (NLISTP LST)
    NIL
    (APPEND (ICODE (CAR LST) PCN PROGRAM)
            (ICOMPILE-PROGRAM-BODY (CDR LST)
                                   (ADD1 PCN)
                                   PROGRAM))).
```

The function **ICODE** takes a Piton instruction, together with the offset (**PCN**) at which it occurs in the body and the entire program (**PROGRAM**) containing the instruction, and generates the i-code for the given instruction. **ICODE** works by using **ICODE1** to generate the i-code and then attaching a label to that block to mark the location at which the corresponding Piton **PC** begins. (The definition of **ICODE** is on page 181). **ICODE1** is simply another big case statement on the opcode of the Piton instruction.

Each Piton instruction has an i-code generator. Our naming convention is that the code for the Piton instruction with name **opcode** is generated by the function named **ICODE-opcode**.

Below we show the generator for the **PUSH-CONSTANT** instruction, i.e., **ICODE-PUSH-CONSTANT**.

Definition.
```
(ICODE-PUSH-CONSTANT INS PCN PROGRAM)
   =
(LIST '(TPUSH_*)
      (COND ((EQUAL (CADR INS) 'PC)
             (TAG 'PC
                  (CONS (NAME PROGRAM) (ADD1 PCN))))
            ((NLISTP (CADR INS))
             (PC (CADR INS) PROGRAM))
            (T (CADR INS))))
```

Observe that the i-code generated for **PUSH-CONSTANT** contains two items. The first is a **TPUSH_*** instruction. The second is the Piton object to be used as immediate data for the **TPUSH_***.

A more interesting instruction, perhaps, is **PUSH-LOCAL**.

Definition.
```
(ICODE-PUSH-LOCAL INS PCN PROGRAM)
   =
(LIST '(MOVE_X_*)
      (TAG 'NAT
           (OFFSET-FROM-CSP (CADR INS) PROGRAM))
      '(ADD_X{N}_CSP)
      '(TPUSH_<X{S}>))
```

This function generates a list of four i-code instructions and data, using **OFFSET-FROM-CSP** to determine the position within the local variables of the local variable to be pushed.

This completes our tour through the compilation phase. The reader is invited to study the i-code generators for each of the Piton instructions.

9.1.3. The Formal Definition of the Link-Assembler

The link-assembler converts an i-state to an m-state by replacing all the i-code instructions in the program segment by bit vectors and all the symbolic data objects (in the stacks, programs, registers and data) by bit vectors. The implementation of the link-assembler is the function

Definition.
```
(I->M I)
   =
(M-STATE (LIST (LINK-WORD (I-PC I) LINK-TABLES (I-WORD-SIZE I))
               (LINK-WORD (I-CFP I) LINK-TABLES (I-WORD-SIZE I))
               (LINK-WORD (I-CSP I) LINK-TABLES (I-WORD-SIZE I))
               (LINK-WORD (I-TSP I) LINK-TABLES (I-WORD-SIZE I))
               (LINK-WORD (I-X I) LINK-TABLES (I-WORD-SIZE I))
               (LINK-WORD (I-Y I) LINK-TABLES (I-WORD-SIZE I))
               (LINK-WORD '(NAT 0) LINK-TABLES (I-WORD-SIZE I))
               (LINK-WORD '(NAT 0) LINK-TABLES (I-WORD-SIZE I)))
         (BOOL-TO-LOGICAL (UNTAG (I-C-FLG I)))
         (BOOL-TO-LOGICAL (UNTAG (I-V-FLG I)))
         (BOOL-TO-LOGICAL (UNTAG (I-N-FLG I)))
         (BOOL-TO-LOGICAL (UNTAG (I-Z-FLG I)))
         (LINK-MEM (I-PROG-SEGMENT I)
                   (I-USR-DATA-SEGMENT I)
                   (I-SYS-DATA-SEGMENT I)
                   (I-LINK-TABLES I)
                   (I-WORD-SIZE I))).
```

The **LIST**-expression in the first argument of the m-state describes the 8 registers. Next come the four condition code registers. Finally, comes the memory. The memory is produced by mapping over the program, user data, and system data segments and appending together the results of linking every word with the function **LINK-WORD**. Before this is done the link tables are computed by **I-LINK-TABLES**.

LINK-WORD operates by first determining whether the word is an i-code instruction or a data word and using the appropriate linker.

Definition.
```
(LINK-WORD X LINK-TABLES WORD-SIZE)
   =
(IF (ICODE-INSTRUCTIONP X)
    (LINK-INSTR-WORD X WORD-SIZE)
    (LINK-DATA-WORD X LINK-TABLES WORD-SIZE))
```

I-code instructions are linked with

Definition.
```
(LINK-INSTR-WORD INS WORD-SIZE)
   =
(MCI (CADR (ASSOC (CAR INS)
                  (LINK-INSTRUCTION-ALIST)))
     WORD-SIZE).
```

The **ASSOC** above looks up the i-code opcode in the table (**LINK-INSTRUCTION-ALIST**) that expands it into an assembly instruction. Then **MCI** (**M**achine **C**ode **I**nstruction) assembles the instruction into a bit vector.

Definition .
```
(MCI INS WORD-SIZE)
   =
(PACK-INSTRUCTION (EXTRACT-OP (CAR INS))
                  (EXTRACT-MOVE-BIT (CAR INS))
                  0
                  (EXTRACT-CVNZ (CADR INS))
                  (EXTRACT-MODE (CADDR INS))
                  (EXTRACT-REG (CADDR INS))
                  (EXTRACT-MODE (CADDDR INS))
                  (EXTRACT-REG (CADDDR INS))
                  WORD-SIZE)
```

If, on the other hand, the word to be linked is a data word, **LINK-DATA-WORD** is used. It case splits on the type of the data word and uses the appropriate mapping to bit vectors.

Definition .
```
(LINK-DATA-WORD X LINK-TABLES WORD-SIZE)
   =
(CASE (TYPE X)
      (NAT       (NAT-TO-BV (UNTAG X) WORD-SIZE))
      (INT       (TC-TO-BV (UNTAG X) WORD-SIZE))
      (BITV      (BITV-TO-BV (UNTAG X) WORD-SIZE))
      (BOOL      (BOOL-TO-BV (UNTAG X) WORD-SIZE))
      (ADDR      (ADDR-TO-BV (UNTAG X)
                             (USR-DATA-LINKS LINK-TABLES)
                             WORD-SIZE))
      (SUBR      (SUBR-TO-BV (UNTAG X)
                             (PROG-LINKS LINK-TABLES)
                             WORD-SIZE))
      (SYS-ADDR  (SYS-ADDR-TO-BV (UNTAG X)
                                 (SYS-DATA-LINKS LINK-TABLES)
                                 WORD-SIZE))
      (PC        (LABEL-TO-BV (UNTAG X)
                              (PROG-LABEL-TABLES LINK-TABLES)
                              WORD-SIZE))
      (IPC       (IPC-TO-BV (UNTAG X)
                            (PROG-LINKS LINK-TABLES)
                            WORD-SIZE))
      (OTHERWISE (NAT-TO-BV 0 WORD-SIZE)))
```

This completes our tour through the link-assembler. The reader is urged to read the code for assembling instructions and mapping each type of data object into bit vectors.

Reader uninterested in pursuing the formal definition of the implementation should skip to page 205.

9.2. Alphabetical Listing of the Implementation

Definition .
```
(ABSOLUTE-ADDRESS ADP LINK-TABLE)
   =
(PLUS (BASE-ADDRESS (ADP-NAME ADP)
                    LINK-TABLE)
      (ADP-OFFSET ADP))
```

Definition.
```
(ADDR-TO-BV ADP USR-DATA-LINKS WORD-SIZE)
   =
(NAT-TO-BV (ABSOLUTE-ADDRESS ADP USR-DATA-LINKS)
           WORD-SIZE)
```

Definition.
```
(ADP-NAME ADP)
   =
(CAR ADP)
```

Definition.
```
(ADP-OFFSET ADP)
   =
(CDR ADP)
```

Definition.
```
(AREA-NAME X)
   =
(ADP-NAME (UNTAG X))
```

Definition.
```
(BASE-ADDRESS NAME LINK-TABLE)
   =
(CDR (ASSOC NAME LINK-TABLE))
```

Definition.
```
(BINDINGS FRAME)
   =
(CAR FRAME)
```

Shell Definition.
Add the shell **BITV** of **2** arguments, with
base function symbol **BTM**,
recognizer function symbol **BITVP**,
accessors **BIT** and **VEC**,
type restrictions **(ONE-OF TRUEP FALSEP)** and **(ONE-OF BITVP)**, and
default function symbols **FALSE** and **BTM**.

Definition.
```
(BITV-TO-BV LST WORD-SIZE)
   =
(IF (ZEROP WORD-SIZE)
    (BTM)
    (V-APPEND (BITV-TO-BV (CDR LST)
                          (SUB1 WORD-SIZE))
              (BITV (IF (EQUAL (CAR LST) 0) F T)
                    (BTM))))
```

Definition.
```
(BOOL-TO-BV B WORD-SIZE)
   =
(IF (EQUAL B 'F)
    (NAT-TO-BV 0 WORD-SIZE)
    (NAT-TO-BV 1 WORD-SIZE))
```

Definition.
```
(BOOL-TO-LOGICAL B)
   =
(IF (EQUAL B 'F) F T)
```

Definition.
```
(BTMP A)
   =
(IF (BITVP A) (EQUAL A (BTM)) T)
```

Definition.
```
(COMPL X)
   =
(IF (BITVP X)
    (IF (EQUAL X (BTM))
        (BTM)
        (BITV (NOT (BIT X)) (COMPL (VEC X))))
    (BTM))
```

Definition.
```
(DEFINITION NAME ALIST)
   =
(ASSOC NAME ALIST)
```

Definition.
```
(DL LAB COMMENT INS)
   =
(LIST 'DL LAB COMMENT INS)
```

Definition.
```
(DL-BLOCK LAB COMMENT BLOCK)
   =
(CONS (DL LAB COMMENT (CAR BLOCK))
      (CDR BLOCK))
```

Definition.
```
(EXP I J)
   =
(IF (ZEROP J)
    1
    (TIMES I (EXP I (SUB1 J))))
```

Definition.
```
(EXTRACT-CVNZ FLG-NAMES)
   =
(PLUS (TIMES (IF (MEMBER 'C FLG-NAMES) 1 0)
             (EXP 2 3))
      (TIMES (IF (MEMBER 'V FLG-NAMES) 1 0)
             (EXP 2 2))
      (TIMES (IF (MEMBER 'N FLG-NAMES) 1 0)
             (EXP 2 1))
      (TIMES (IF (MEMBER 'Z FLG-NAMES) 1 0)
             (EXP 2 0)))
```

Definition.
```
(EXTRACT-MODE REG-SPEC)
   =
(COND ((LITATOM REG-SPEC) 0)
      ((EQUAL (CDR REG-SPEC) NIL) 1)
      ((EQUAL (CAR REG-SPEC) -1) 2)
      (T 3))
```

Definition.
```
(EXTRACT-MOVE-BIT OPCODE)
   =
(IF (MEMBER OPCODE
            '(MOVE MOVE-NC MOVE-C MOVE-NV MOVE-V MOVE-NZ MOVE-Z MOVE-NN
                   MOVE-N))
    1 0)
```

Definition.
```
(EXTRACT-OP OPCODE)
   =
(CADR (ASSOC OPCODE
             '((INCR 1)
               (ADDC 2)
               (ADD 3)
               (NEG 4)
               (DECR 5)
               (SUBB 6)
               (SUB 7)
               (ROR 8)
               (ASR 9)
               (LSR 10)
               (XOR 11)
               (OR 12)
               (AND 13)
               (NOT 14)
               (MOVE 15)
               (MOVE-NC 0)
               (MOVE-C 1)
               (MOVE-NV 2)
               (MOVE-V 3)
               (MOVE-NZ 4)
               (MOVE-Z 5)
               (MOVE-NN 6)
               (MOVE-N 7))))
```

Definition.
```
(EXTRACT-REG REG-SPEC)
   =
(CADR (ASSOC (EXTRACT-REG1 REG-SPEC)
             '((PC 0)
               (CFP 1)
               (CSP 2)
               (TSP 3)
               (X 4)
               (Y 5))))
```

Definition.
```
(EXTRACT-REG1 REG-SPEC)
   =
(COND ((LITATOM REG-SPEC) REG-SPEC)
      ((EQUAL (CDR REG-SPEC) NIL)
       (CAR REG-SPEC))
      ((EQUAL (CAR REG-SPEC) -1)
       (CADR REG-SPEC))
      (T (CAR REG-SPEC)))
```

Definition.
```
(FIND-LABEL X LST)
   =
(COND ((NLISTP LST) 0)
      ((AND (LABELLEDP (CAR LST))
            (EQUAL X (CADAR LST)))
       0)
      (T (ADD1 (FIND-LABEL X (CDR LST)))))
```

Definition.
```
(FIND-POSITION-OF-VAR VAR LST)
   =
(COND ((NLISTP LST) 0)
      ((EQUAL VAR (CAR LST)) 0)
      (T (ADD1 (FIND-POSITION-OF-VAR VAR
                                     (CDR LST)))))
```

Definition.
```
(FORMAL-VARS D)
   =
(CADR D)
```

Definition.
```
(GENERATE-POSTLUDE PROGRAM)
   =
(LIST (DL (CONS (NAME PROGRAM)
                (LENGTH (PROGRAM-BODY PROGRAM)))
          '(POSTLUDE)
          '(MOVE_CSP_CFP))
      '(CPOP_CFP)
      '(CPOP_PC))
```

Definition.
```
(GENERATE-PRELUDE PROGRAM)
   =
(APPEND (LIST (DL (CONS (NAME PROGRAM) '(PRELUDE))
                  '(PRELUDE)
                  '(CPUSH_CFP))
              '(MOVE_CFP_CSP))
        (APPEND (GENERATE-PRELUDE1 (REVERSE (TEMP-VAR-DCLS PROGRAM)))
                (GENERATE-PRELUDE2 (FORMAL-VARS PROGRAM))))
```

Definition.
```
(GENERATE-PRELUDE1 TEMP-VAR-DCLS)
   =
(IF (NLISTP TEMP-VAR-DCLS)
    NIL
    (CONS '(CPUSH_*)
          (CONS (CADAR TEMP-VAR-DCLS)
                (GENERATE-PRELUDE1 (CDR TEMP-VAR-DCLS)))))
```

Definition.
```
(GENERATE-PRELUDE2 FORMAL-VARS)
   =
(IF (NLISTP FORMAL-VARS)
    NIL
    (CONS '(CPUSH_<TSP>+)
          (GENERATE-PRELUDE2 (CDR FORMAL-VARS))))
```

Definition.
```
(I->M I)
   =
(M-STATE (LIST (LINK-WORD (I-PC I) LINK-TABLES (I-WORD-SIZE I))
               (LINK-WORD (I-CFP I) LINK-TABLES (I-WORD-SIZE I))
               (LINK-WORD (I-CSP I) LINK-TABLES (I-WORD-SIZE I))
               (LINK-WORD (I-TSP I) LINK-TABLES (I-WORD-SIZE I))
               (LINK-WORD (I-X I) LINK-TABLES (I-WORD-SIZE I))
               (LINK-WORD (I-Y I) LINK-TABLES (I-WORD-SIZE I))
               (LINK-WORD '(NAT 0) LINK-TABLES (I-WORD-SIZE I))
               (LINK-WORD '(NAT 0) LINK-TABLES (I-WORD-SIZE I)))
         (BOOL-TO-LOGICAL (UNTAG (I-C-FLG I)))
         (BOOL-TO-LOGICAL (UNTAG (I-V-FLG I)))
         (BOOL-TO-LOGICAL (UNTAG (I-N-FLG I)))
         (BOOL-TO-LOGICAL (UNTAG (I-Z-FLG I)))
         (LINK-MEM (I-PROG-SEGMENT I)
                   (I-USR-DATA-SEGMENT I)
                   (I-SYS-DATA-SEGMENT I)
                   (I-LINK-TABLES I)
                   (I-WORD-SIZE I)))
```

Definition.
```
(I-LINK-TABLES I)
   =
(LIST
 (LINK-TABLE-FOR-SEGMENT (I-PROG-SEGMENT I)
                         0)
 (LINK-TABLE-FOR-PROG-LABELS (I-PROG-SEGMENT I)
                             0)
 (LINK-TABLE-FOR-SEGMENT (I-USR-DATA-SEGMENT I)
                         (SEGMENT-LENGTH (I-PROG-SEGMENT I)))
 (LINK-TABLE-FOR-SEGMENT (I-SYS-DATA-SEGMENT I)
                         (PLUS
                          (SEGMENT-LENGTH (I-PROG-SEGMENT I))
                          (SEGMENT-LENGTH (I-USR-DATA-SEGMENT I)))))
```

Shell Definition.
Add the shell **I-STATE** of **15** arguments, with
recognizer function symbol **I-STATEP**, and
accessors **I-PC**, **I-CFP**, **I-CSP**, **I-TSP**, **I-X**, **I-Y**,
I-C-FLG, **I-V-FLG**, **I-N-FLG**, **I-Z-FLG**,
I-PROG-SEGMENT, **I-USR-DATA-SEGMENT**, **I-SYS-DATA-SEGMENT**,
I-WORD-SIZE and **I-PSW**.

Definition.
```
(ICODE INS PCN PROGRAM)
   =
(DL-BLOCK (CONS (NAME PROGRAM) PCN)
          INS
          (ICODE1 (UNLABEL INS) PCN PROGRAM))
```

Definition.
```
(ICODE-ADD-ADDR INS PCN PROGRAM)
   =
'((TPOP_X) (ADD_<TSP>{A}_X{N}))
```

Definition.
```
(ICODE-ADD-INT INS PCN PROGRAM)
  =
'((TPOP_X) (ADD_<TSP>{I}_X{I}))
```

Definition.
```
(ICODE-ADD-INT-WITH-CARRY INS PCN PROGRAM)
  =
'((TPOP_X)
  (TPOP_Y)
  (ASR_<C>_<TSP>_<TSP>{B})
  (ADDC_<V>_X{I}_Y{I})
  (MOVE-V_<TSP>_*)
  (BOOL T)
  (TPUSH_X))
```

Definition.
```
(ICODE-ADD-NAT INS PCN PROGRAM)
  =
'((TPOP_X) (ADD_<TSP>{N}_X{N}))
```

Definition.
```
(ICODE-ADD-NAT-WITH-CARRY INS PCN PROGRAM)
  =
'((TPOP_X)
  (TPOP_Y)
  (ASR_<C>_<TSP>_<TSP>{B})
  (ADDC_<C>_X{N}_Y{N})
  (MOVE-C_<TSP>_*)
  (BOOL T)
  (TPUSH_X))
```

Definition.
```
(ICODE-ADD1-INT INS PCN PROGRAM)
  =
'((INCR_<TSP>_<TSP>{I}))
```

Definition.
```
(ICODE-ADD1-NAT INS PCN PROGRAM)
  =
'((INCR_<TSP>_<TSP>{N}))
```

Definition.
```
(ICODE-AND-BITV INS PCN PROGRAM)
  =
'((TPOP_X) (AND_<TSP>{V}_X{V}))
```

Definition.
```
(ICODE-AND-BOOL INS PCN PROGRAM)
  =
'((TPOP_X) (AND_<TSP>{B}_X{B}))
```

Definition.

```
(ICODE-CALL INS PCN PROGRAM)
   =
(LIST '(CPUSH_*)
      (TAG 'PC
           (CONS (NAME PROGRAM) (ADD1 PCN)))
      '(JUMP_*)
      (TAG 'PC
           (CONS (CADR INS) '(PRELUDE))))
```

Definition.

```
(ICODE-DEPOSIT INS PCN PROGRAM)
   =
'((TPOP_X) (TPOP_<X{A}>))
```

Definition.

```
(ICODE-DEPOSIT-TEMP-STK INS PCN PROGRAM)
   =
'((TPOP_Y)
  (INCR_Y_Y{N})
  (MOVE_X_*)
  (SYS-ADDR (EMPTY-TEMP-STK-ADDR . 0))
  (MOVE_X_<X{S}>)
  (SUB_X{S}_Y{N})
  (TPOP_<X{S}>))
```

Definition.

```
(ICODE-DIV2-NAT INS PCN PROGRAM)
   =
'((TPOP_<C>_X)
  (LSR_<C>_X_X{N})
  (TPUSH_X)
  (TPUSH_*)
  (NAT 0)
  (MOVE-C_<TSP>_*)
  (NAT 1))
```

Definition.

```
(ICODE-EQ INS PCN PROGRAM)
   =
'((TPOP_X)
  (XOR_<Z>_<TSP>_X)
  (XOR_<TSP>_<TSP>)
  (MOVE-Z_<TSP>_*)
  (BOOL T))
```

Definition.

```
(ICODE-FETCH INS PCN PROGRAM)
   =
'((TPOP_X) (TPUSH_<X{A}>))
```

Definition.

```
(ICODE-FETCH-TEMP-STK INS PCN PROGRAM)
   =
'((TPOP_Y)
  (INCR_Y_Y{N})
  (MOVE_X_*)
  (SYS-ADDR (EMPTY-TEMP-STK-ADDR . 0))
  (MOVE_X_<X{S}>)
  (SUB_X{S}_Y{N})
  (TPUSH_<X{S}>))
```

Definition.

```
(ICODE-INSTRUCTIONP INS)
   =
(EQUAL (CDR INS) NIL)
```

Definition.

```
(ICODE-INT-TO-NAT INS PCN PROGRAM)
   =
'((INT-TO-NAT))
```

Definition.

```
(ICODE-JUMP INS PCN PROGRAM)
   =
(LIST '(JUMP_*)
      (PC (CADR INS) PROGRAM))
```

Definition.

```
(ICODE-JUMP-CASE INS PCN PROGRAM)
   =
(APPEND '((TPOP_X) (ADD_X_X{N}) (ADD_PC_X{N}))
        (JUMP_*-LST (CDR INS) PROGRAM))
```

Definition.

```
(ICODE-JUMP-IF-TEMP-STK-EMPTY INS PCN PROGRAM)
   =
(LIST '(MOVE_Y_TSP)
      '(MOVE_X_*)
      '(SYS-ADDR (EMPTY-TEMP-STK-ADDR . 0))
      '(MOVE_X_<X{S}>)
      '(SUB_<Z>_X{S}_Y{S})
      '(MOVE_X_*)
      (PC (CADR INS) PROGRAM)
      '(JUMP-Z_X))
```

Definition.

```
(ICODE-JUMP-IF-TEMP-STK-FULL INS PCN PROGRAM)
   =
(LIST '(MOVE_X_TSP)
      '(MOVE_Y_*)
      '(SYS-ADDR (FULL-TEMP-STK-ADDR . 0))
      '(MOVE_Y_<Y{S}>)
      '(SUB_<Z>_X{S}_Y{S})
      '(MOVE_X_*)
      (PC (CADR INS) PROGRAM)
      '(JUMP-Z_X))
```

Definition.
```
(ICODE-LOCN INS PCN PROGRAM)
   =
(LIST '(MOVE_X_*)
      (TAG 'NAT
           (OFFSET-FROM-CSP (CADR INS) PROGRAM))
      '(ADD_X{N}_CSP)
      '(MOVE_X_<X{S}>)
      '(ADD_X{N}_CSP)
      '(TPUSH_<X{S}>))
```

Definition.
```
(ICODE-LSH-BITV INS PCN PROGRAM)
   =
'((ADD_<TSP>_<TSP>{V}))
```

Definition.
```
(ICODE-LT-ADDR INS PCN PROGRAM)
   =
'((TPOP_X)
  (SUB_<C>_<TSP>{A}_X{A})
  (XOR_<TSP>_<TSP>)
  (MOVE-C_<TSP>_*)
  (BOOL T))
```

Definition.
```
(ICODE-LT-INT INS PCN PROGRAM)
   =
'((TPOP_X)
  (SUB_<NV>_<TSP>{I}_X{I})
  (MOVE_<TSP>_*)
  (BOOL F)
  (MOVE-V_<TSP>_*)
  (BOOL T)
  (MOVE_X_*)
  (BOOL F)
  (MOVE-N_X_*)
  (BOOL T)
  (XOR_<TSP>{B}_X{B}))
```

Definition.
```
(ICODE-LT-NAT INS PCN PROGRAM)
   =
'((TPOP_X)
  (SUB_<C>_<TSP>{N}_X{N})
  (XOR_<TSP>_<TSP>)
  (MOVE-C_<TSP>_*)
  (BOOL T))
```

Definition.
```
(ICODE-MULT2-NAT INS PCN PROGRAM)
   =
'((ADD_<TSP>_<TSP>{N}))
```

Definition.
```
(ICODE-MULT2-NAT-WITH-CARRY-OUT INS PCN PROGRAM)
   =
'((TPOP_X)
  (ADD_<C>_X_X{N})
  (TPUSH_*)
  (BOOL F)
  (MOVE-C_<TSP>_*)
  (BOOL T)
  (TPUSH_X))
```

Definition.
```
(ICODE-NEG-INT INS PCN PROGRAM)
   =
'((NEG_<TSP>_<TSP>{I}))
```

Definition.
```
(ICODE-NO-OP INS PCN PROGRAM)
   =
'((MOVE_X_X))
```

Definition.
```
(ICODE-NOT-BITV INS PCN PROGRAM)
   =
'((NOT_<TSP>_<TSP>{V}))
```

Definition.
```
(ICODE-NOT-BOOL INS PCN PROGRAM)
   =
'((XOR_<TSP>{B}_*{B}) (BOOL T))
```

Definition.
```
(ICODE-OR-BITV INS PCN PROGRAM)
   =
'((TPOP_X) (OR_<TSP>{V}_X{V}))
```

Definition.
```
(ICODE-OR-BOOL INS PCN PROGRAM)
   =
'((TPOP_X) (OR_<TSP>{B}_X{B}))
```

Definition.
```
(ICODE-POP INS PCN PROGRAM)
   =
'((TPOP_X))
```

Definition.
```
(ICODE-POP* INS PCN PROGRAM)
   =
(LIST '(ADD_TSP_*{N})
      (TAG 'NAT (CADR INS)))
```

Definition.
```
(ICODE-POP-CALL INS PCN PROGRAM)
   =
(LIST '(TPOP_X)
      '(CPUSH_*)
      (TAG 'PC
           (CONS (NAME PROGRAM) (ADD1 PCN)))
     '(JUMP_X{SUBR}))
```

Definition.
```
(ICODE-POP-GLOBAL INS PCN PROGRAM)
   =
(LIST '(MOVE_X_*)
      (TAG 'ADDR (CONS (CADR INS) 0))
      '(TPOP_<X{A}>))
```

Definition.
```
(ICODE-POP-LOCAL INS PCN PROGRAM)
   =
(LIST '(MOVE_X_*)
      (TAG 'NAT
           (OFFSET-FROM-CSP (CADR INS) PROGRAM))
      '(ADD_X{N}_CSP)
      '(TPOP_<X{S}>))
```

Definition.
```
(ICODE-POP-LOCN INS PCN PROGRAM)
   =
(LIST '(MOVE_X_*)
      (TAG 'NAT
           (OFFSET-FROM-CSP (CADR INS) PROGRAM))
      '(ADD_X{N}_CSP)
      '(MOVE_X_<X{S}>)
      '(ADD_X{N}_CSP)
      '(TPOP_<X{S}>))
```

Definition.
```
(ICODE-POPJ INS PCN PROGRAM)
   =
'((TPOP_PC))
```

Definition.
```
(ICODE-POPN INS PCN PROGRAM)
   =
'((TPOP_X) (ADD_TSP_X{N}))
```

Definition.
```
(ICODE-PUSH-CONSTANT INS PCN PROGRAM)
   =
(LIST '(TPUSH_*)
      (COND ((EQUAL (CADR INS) 'PC)
              (TAG 'PC
                   (CONS (NAME PROGRAM) (ADD1 PCN))))
            ((NLISTP (CADR INS))
              (PC (CADR INS) PROGRAM))
            (T (CADR INS))))
```

Definition.
```
(ICODE-PUSH-CTRL-STK-FREE-SIZE INS PCN PROGRAM)
   =
'((MOVE_X_*)
  (SYS-ADDR (FULL-CTRL-STK-ADDR . 0))
  (MOVE_X_<X{S}>)
  (TPUSH_CSP)
  (SUB_<TSP>{S}_X{S}))
```

Definition.
```
(ICODE-PUSH-GLOBAL INS PCN PROGRAM)
   =
(LIST '(MOVE_X_*)
      (TAG 'ADDR (CONS (CADR INS) 0))
      '(TPUSH_<X{A}>))
```

Definition.
```
(ICODE-PUSH-LOCAL INS PCN PROGRAM)
   =
(LIST '(MOVE_X_*)
      (TAG 'NAT
           (OFFSET-FROM-CSP (CADR INS) PROGRAM))
      '(ADD_X{N}_CSP)
      '(TPUSH_<X{S}>))
```

Definition.
```
(ICODE-PUSH-TEMP-STK-FREE-SIZE INS PCN PROGRAM)
   =
'((MOVE_X_*)
  (SYS-ADDR (FULL-TEMP-STK-ADDR . 0))
  (MOVE_X_<X{S}>)
  (TPUSH_TSP)
  (SUB_<TSP>{S}_X{S}))
```

Definition.
```
(ICODE-PUSH-TEMP-STK-INDEX INS PCN PROGRAM)
   =
(LIST '(MOVE_Y_TSP)
      '(MOVE_X_*)
      '(SYS-ADDR (EMPTY-TEMP-STK-ADDR . 0))
      '(MOVE_X_<X{S}>)
      '(SUB_<Z>_X{S}_Y{S})
      '(TPUSH_X)
      '(MOVE_X_*)
      (TAG 'NAT (ADD1 (CADR INS)))
      '(SUB_<TSP>{N}_X{N}))
```

Definition.
```
(ICODE-PUSHJ INS PCN PROGRAM)
   =
(LIST '(TPUSH_*)
      (TAG 'PC
           (CONS (NAME PROGRAM) (ADD1 PCN)))
      '(JUMP_*)
      (PC (CADR INS) PROGRAM))
```

Definition.
```
(ICODE-RET INS PCN PROGRAM)
   =
(LIST '(JUMP_*)
      (TAG 'PC
           (CONS (NAME PROGRAM)
                 (LENGTH (PROGRAM-BODY PROGRAM)))))
```

Definition.
```
(ICODE-RSH-BITV INS PCN PROGRAM)
   =
'((LSR_<TSP>_<TSP>{V}))
```

Definition.
```
(ICODE-SET-GLOBAL INS PCN PROGRAM)
   =
(LIST '(MOVE_X_*)
      (TAG 'ADDR (CONS (CADR INS) 0))
      '(MOVE_<X{A}>_<TSP>))
```

Definition.
```
(ICODE-SET-LOCAL INS PCN PROGRAM)
   =
(LIST '(MOVE_X_*)
      (TAG 'NAT
           (OFFSET-FROM-CSP (CADR INS) PROGRAM))
      '(ADD_X{N}_CSP)
      '(MOVE_<X{S}>_<TSP>))
```

Definition.
```
(ICODE-SUB-ADDR INS PCN PROGRAM)
   =
'((TPOP_X) (SUB_<TSP>{A}_X{N}))
```

Definition.
```
(ICODE-SUB-INT INS PCN PROGRAM)
   =
'((TPOP_X) (SUB_<TSP>{I}_X{I}))
```

Definition.
```
(ICODE-SUB-INT-WITH-CARRY INS PCN PROGRAM)
   =
'((TPOP_Y)
  (TPOP_X)
  (ASR_<C>_<TSP>_<TSP>{B})
  (SUBB_<V>_X{I}_Y{I})
  (MOVE-V_<TSP>_*)
  (BOOL T)
  (TPUSH_X))
```

Definition.
```
(ICODE-SUB-NAT INS PCN PROGRAM)
   =
'((TPOP_X) (SUB_<TSP>{N}_X{N}))
```

Definition.
```
(ICODE-SUB-NAT-WITH-CARRY INS PCN PROGRAM)
   =
'((TPOP_Y)
  (TPOP_X)
  (ASR_<C>_<TSP>_<TSP>{B})
  (SUBB_<C>_X{N}_Y{N})
  (MOVE-C_<TSP>_*)
  (BOOL T)
  (TPUSH_X))
```

Definition.
```
(ICODE-SUB1-INT INS PCN PROGRAM)
   =
'((DECR_<TSP>_<TSP>{I}))
```

Definition.
```
(ICODE-SUB1-NAT INS PCN PROGRAM)
   =
'((DECR_<TSP>_<TSP>{N}))
```

Definition.
```
(ICODE-TEST-BITV-AND-JUMP INS PCN PROGRAM)
   =
(IF (EQUAL (CADR INS) 'ALL-ZERO)
    (LIST '(TPOP{V}_<Z>_Y)
          '(MOVE_X_*)
          (PC (CADDR INS) PROGRAM)
          '(JUMP-Z_X))
    (LIST '(TPOP{V}_<Z>_Y)
          '(MOVE_X_*)
          (PC (CADDR INS) PROGRAM)
          '(JUMP-NZ_X)))
```

Definition.
```
(ICODE-TEST-BOOL-AND-JUMP INS PCN PROGRAM)
   =
(IF (EQUAL (CADR INS) 'T)
    (LIST '(TPOP{B}_<Z>_Y)
          '(MOVE_X_*)
          (PC (CADDR INS) PROGRAM)
          '(JUMP-NZ_X))
    (LIST '(TPOP{B}_<Z>_Y)
          '(MOVE_X_*)
          (PC (CADDR INS) PROGRAM)
          '(JUMP-Z_X)))
```

Definition.

```
(ICODE-TEST-INT-AND-JUMP INS PCN PROGRAM)
   =
(CASE (CAR (CDR INS))
      (ZERO      (LIST '(TPOP{I}_<ZN>_Y)
                       '(MOVE_X_*)
                       (PC (CADDR INS) PROGRAM)
                       '(JUMP-Z_X)))
      (NOT-ZERO  (LIST '(TPOP{I}_<ZN>_Y)
                       '(MOVE_X_*)
                       (PC (CADDR INS) PROGRAM)
                       '(JUMP-NZ_X)))
      (NEG       (LIST '(TPOP{I}_<ZN>_Y)
                       '(MOVE_X_*)
                       (PC (CADDR INS) PROGRAM)
                       '(JUMP-N_X)))
      (NOT-NEG   (LIST '(TPOP{I}_<ZN>_Y)
                       '(MOVE_X_*)
                       (PC (CADDR INS) PROGRAM)
                       '(JUMP-NN_X)))
      (POS       (LIST '(TPOP{I}_<ZN>_Y)
                       '(MOVE_X_*)
                       (TAG 'PC
                            (CONS (NAME PROGRAM) (ADD1 PCN)))
                       '(JUMP-N_X)
                       '(JUMP-Z_X)
                       '(JUMP_*)
                       (PC (CADDR INS) PROGRAM)))
      (OTHERWISE (LIST '(TPOP{I}_<ZN>_Y)
                       '(MOVE_X_*)
                       (PC (CADDR INS) PROGRAM)
                       '(JUMP-N_X)
                       '(JUMP-Z_X))))
```

Definition.

```
(ICODE-TEST-NAT-AND-JUMP INS PCN PROGRAM)
   =
(IF (EQUAL (CADR INS) 'ZERO)
    (LIST '(TPOP{N}_<Z>_Y)
          '(MOVE_X_*)
          (PC (CADDR INS) PROGRAM)
          '(JUMP-Z_X))
    (LIST '(TPOP{N}_<Z>_Y)
          '(MOVE_X_*)
          (PC (CADDR INS) PROGRAM)
          '(JUMP-NZ_X)))
```

Definition.

```
(ICODE-XOR-BITV INS PCN PROGRAM)
   =
'((TPOP_X) (XOR_<TSP>{V}_X{V}))
```

Definition.

```
(ICODE1 INS PCN PROG)
   =
(CASE (CAR INS)
      (CALL                (ICODE-CALL INS PCN PROG))
      (RET                 (ICODE-RET INS PCN PROG))
      (LOCN                (ICODE-LOCN INS PCN PROG))
      (PUSH-CONSTANT       (ICODE-PUSH-CONSTANT INS PCN PROG))
      (PUSH-LOCAL          (ICODE-PUSH-LOCAL INS PCN PROG))
      (PUSH-GLOBAL         (ICODE-PUSH-GLOBAL INS PCN PROG))

      (PUSH-CTRL-STK-FREE-SIZE
                           (ICODE-PUSH-CTRL-STK-FREE-SIZE INS PCN PROG))
      (PUSH-TEMP-STK-FREE-SIZE
                           (ICODE-PUSH-TEMP-STK-FREE-SIZE INS PCN PROG))
      (PUSH-TEMP-STK-INDEX
                           (ICODE-PUSH-TEMP-STK-INDEX INS PCN PROG))
      (JUMP-IF-TEMP-STK-FULL
                           (ICODE-JUMP-IF-TEMP-STK-FULL INS PCN PROG))
      (JUMP-IF-TEMP-STK-EMPTY
                           (ICODE-JUMP-IF-TEMP-STK-EMPTY INS PCN PROG))
      (POP                 (ICODE-POP INS PCN PROG))
      (POP*                (ICODE-POP* INS PCN PROG))
      (POPN                (ICODE-POPN INS PCN PROG))
      (POP-LOCAL           (ICODE-POP-LOCAL INS PCN PROG))
      (POP-GLOBAL          (ICODE-POP-GLOBAL INS PCN PROG))
      (POP-LOCN            (ICODE-POP-LOCN INS PCN PROG))
      (POP-CALL            (ICODE-POP-CALL INS PCN PROG))
      (FETCH-TEMP-STK      (ICODE-FETCH-TEMP-STK INS PCN PROG))
      (DEPOSIT-TEMP-STK    (ICODE-DEPOSIT-TEMP-STK INS PCN PROG))
      (JUMP                (ICODE-JUMP INS PCN PROG))
      (JUMP-CASE           (ICODE-JUMP-CASE INS PCN PROG))
      (PUSHJ               (ICODE-PUSHJ INS PCN PROG))
      (POPJ                (ICODE-POPJ INS PCN PROG))
      (SET-LOCAL           (ICODE-SET-LOCAL INS PCN PROG))
      (SET-GLOBAL          (ICODE-SET-GLOBAL INS PCN PROG))
      (TEST-NAT-AND-JUMP   (ICODE-TEST-NAT-AND-JUMP INS PCN PROG))
      (TEST-INT-AND-JUMP   (ICODE-TEST-INT-AND-JUMP INS PCN PROG))
      (TEST-BOOL-AND-JUMP  (ICODE-TEST-BOOL-AND-JUMP INS PCN PROG))
      (TEST-BITV-AND-JUMP  (ICODE-TEST-BITV-AND-JUMP INS PCN PROG))
      (NO-OP               (ICODE-NO-OP INS PCN PROG))
      (ADD-ADDR            (ICODE-ADD-ADDR INS PCN PROG))
      (SUB-ADDR            (ICODE-SUB-ADDR INS PCN PROG))
      (EQ                  (ICODE-EQ INS PCN PROG))
      (LT-ADDR             (ICODE-LT-ADDR INS PCN PROG))
      (FETCH               (ICODE-FETCH INS PCN PROG))
      (DEPOSIT             (ICODE-DEPOSIT INS PCN PROG))
      (ADD-INT             (ICODE-ADD-INT INS PCN PROG))
      (ADD-INT-WITH-CARRY  (ICODE-ADD-INT-WITH-CARRY INS PCN PROG))
      (ADD1-INT            (ICODE-ADD1-INT INS PCN PROG))
      (SUB-INT             (ICODE-SUB-INT INS PCN PROG))
      (SUB-INT-WITH-CARR   (ICODE-SUB-INT-WITH-CARRY INS PCN PROG))
      (SUB1-INT            (ICODE-SUB1-INT INS PCN PROG))
      (NEG-INT             (ICODE-NEG-INT INS PCN PROG))
      (LT-INT              (ICODE-LT-INT INS PCN PROG))
      (INT-TO-NAT          (ICODE-INT-TO-NAT INS PCN PROG))
      (ADD-NAT             (ICODE-ADD-NAT INS PCN PROG))
```

```
      (ADD-NAT-WITH-CARRY (ICODE-ADD-NAT-WITH-CARRY INS PCN PROG))
      (ADD1-NAT           (ICODE-ADD1-NAT INS PCN PROG))
      (SUB-NAT            (ICODE-SUB-NAT INS PCN PROG))
      (SUB-NAT-WITH-CARRY (ICODE-SUB-NAT-WITH-CARRY INS PCN PROG))
      (SUB1-NAT           (ICODE-SUB1-NAT INS PCN PROG))
      (LT-NAT             (ICODE-LT-NAT INS PCN PROG))
      (MULT2-NAT          (ICODE-MULT2-NAT INS PCN PROG))
      (MULT2-NAT-WITH-CARRY-OUT
                          (ICODE-MULT2-NAT-WITH-CARRY-OUT INS PCN PROG))
      (DIV2-NAT           (ICODE-DIV2-NAT INS PCN PROG))
      (OR-BITV            (ICODE-OR-BITV INS PCN PROG))
      (AND-BITV           (ICODE-AND-BITV INS PCN PROG))
      (NOT-BITV           (ICODE-NOT-BITV INS PCN PROG))
      (XOR-BITV           (ICODE-XOR-BITV INS PCN PROG))
      (RSH-BITV           (ICODE-RSH-BITV INS PCN PROG))
      (LSH-BITV           (ICODE-LSH-BITV INS PCN PROG))
      (OR-BOOL            (ICODE-OR-BOOL INS PCN PROG))
      (AND-BOOL           (ICODE-AND-BOOL INS PCN PROG))
      (NOT-BOOL           (ICODE-NOT-BOOL INS PCN PROG))
      (OTHERWISE          '((ERROR))))
```

Definition.

```
(ICOMPILE PROGRAMS)
   =
(IF (NLISTP PROGRAMS)
    NIL
    (CONS (ICOMPILE-PROGRAM (CAR PROGRAMS))
          (ICOMPILE (CDR PROGRAMS))))
```

Definition.

```
(ICOMPILE-PROGRAM PROGRAM)
   =
(CONS (NAME PROGRAM)
      (APPEND (GENERATE-PRELUDE PROGRAM)
              (APPEND (ICOMPILE-PROGRAM-BODY (PROGRAM-BODY PROGRAM)
                                             0 PROGRAM)
                      (GENERATE-POSTLUDE PROGRAM))))
```

Definition.

```
(ICOMPILE-PROGRAM-BODY LST PCN PROGRAM)
   =
(IF (NLISTP LST)
    NIL
    (APPEND (ICODE (CAR LST) PCN PROGRAM)
            (ICOMPILE-PROGRAM-BODY (CDR LST)
                                   (ADD1 PCN)
                                   PROGRAM)))
```

Definition.

```
(INCR C X)
   =
(IF (BITVP X)
    (IF (EQUAL X (BTM))
        (BTM)
        (BITV (XOR C (BIT X))
              (INCR (AND C (BIT X)) (VEC X))))
    (BTM))
```

Definition.
```
(IPC-TO-BV PCPP PROG-LINKS WORD-SIZE)
   =
(NAT-TO-BV (ABSOLUTE-ADDRESS PCPP PROG-LINKS)
           WORD-SIZE)
```

Definition.
```
(JUMP_*-LST LST PROGRAM)
   =
(IF (NLISTP LST)
    NIL
    (CONS '(JUMP_*)
          (CONS (PC (CAR LST) PROGRAM)
                (JUMP_*-LST (CDR LST) PROGRAM))))
```

Definition.
```
(LABEL-ADDRESS LABEL PROG-LABEL-TABLES)
   =
(BASE-ADDRESS LABEL
              (LABEL-LINKS LABEL PROG-LABEL-TABLES))
```

Definition.
```
(LABEL-LINKS LABEL PROG-LABEL-TABLES)
   =
(CDR (ASSOC (ADP-NAME LABEL)
            PROG-LABEL-TABLES))
```

Definition.
```
(LABEL-TO-BV ILAB PROG-LABEL-TABLES WORD-SIZE)
   =
(NAT-TO-BV (LABEL-ADDRESS ILAB PROG-LABEL-TABLES)
           WORD-SIZE)
```

Definition.
```
(LABELLEDP X)
   =
(EQUAL (CAR X) 'DL)
```

Definition.
```
(LENGTH X)
   =
(IF (NLISTP X)
    0
    (ADD1 (LENGTH (CDR X))))
```

Definition.
```
(LINK-AREA LST LINK-TABLES WORD-SIZE)
   =
(IF (NLISTP LST)
    NIL
    (CONS (LINK-WORD (UNLABEL (CAR LST))
                     LINK-TABLES WORD-SIZE)
          (LINK-AREA (CDR LST)
                     LINK-TABLES WORD-SIZE)))
```

Definition.
```
(LINK-DATA-WORD X LINK-TABLES WORD-SIZE)
   =
(CASE (TYPE X)
      (NAT       (NAT-TO-BV (UNTAG X) WORD-SIZE))
      (INT       (TC-TO-BV (UNTAG X) WORD-SIZE))
      (BITV      (BITV-TO-BV (UNTAG X) WORD-SIZE))
      (BOOL      (BOOL-TO-BV (UNTAG X) WORD-SIZE))
      (ADDR      (ADDR-TO-BV (UNTAG X)
                             (USR-DATA-LINKS LINK-TABLES)
                             WORD-SIZE))
      (SUBR      (SUBR-TO-BV (UNTAG X)
                             (PROG-LINKS LINK-TABLES)
                             WORD-SIZE))
      (SYS-ADDR  (SYS-ADDR-TO-BV (UNTAG X)
                                 (SYS-DATA-LINKS LINK-TABLES)
                                 WORD-SIZE))
      (PC        (LABEL-TO-BV (UNTAG X)
                              (PROG-LABEL-TABLES LINK-TABLES)
                              WORD-SIZE))
      (IPC       (IPC-TO-BV (UNTAG X)
                            (PROG-LINKS LINK-TABLES)
                            WORD-SIZE))
      (OTHERWISE (NAT-TO-BV 0 WORD-SIZE)))
```

Definition.
```
(LINK-INSTR-WORD INS WORD-SIZE)
   =
(MCI (CADR (ASSOC (CAR INS)
                  (LINK-INSTRUCTION-ALIST)))
     WORD-SIZE)
```

Definition.
```
(LINK-INSTRUCTION-ALIST)
   =
'((ADD_<C>_X_X{N} (ADD (C) X X))
  (ADD_<TSP>_<TSP>{V} (ADD NIL (TSP) (TSP)))
  (ADD_<TSP>_<TSP>{N} (ADD NIL (TSP) (TSP)))
  (ADD_<TSP>{A}_X{N} (ADD NIL (TSP) X))
  (ADD_TSP_*{N} (ADD NIL TSP (PC 1)))
  (ADD_TSP_X{N} (ADD NIL TSP X))
  (ADD_<TSP>{I}_X{I} (ADD NIL (TSP) X))
  (ADD_<TSP>{N}_X{N} (ADD NIL (TSP) X))
  (ADD_PC_X{N} (ADD NIL PC X))
  (ADD_X_X{N} (ADD NIL X X))
  (ADD_X{N}_CSP (ADD NIL X CSP))
  (ADDC_<C>_X{N}_Y{N} (ADDC (C) X Y))
  (ADDC_<V>_X{I}_Y{I} (ADDC (V) X Y))
  (AND_<TSP>{V}_X{V} (AND NIL (TSP) X))
  (AND_<TSP>{B}_X{B} (AND NIL (TSP) X))
  (ASR_<C>_<TSP>_<TSP>{B} (ASR (C) (TSP) (TSP)))
  (CPOP_CFP (MOVE NIL CFP (CSP 1)))
  (CPOP_PC (MOVE NIL PC (CSP 1)))
  (CPUSH_* (MOVE NIL (-1 CSP) (PC 1)))
  (CPUSH_<TSP>+ (MOVE NIL (-1 CSP) (TSP 1)))
  (CPUSH_CFP (MOVE NIL (-1 CSP) CFP))
  (DECR_<TSP>_<TSP>{I} (DECR NIL (TSP) (TSP)))
  (DECR_<TSP>_<TSP>{N} (DECR NIL (TSP) (TSP)))
```

```
(INCR_<TSP>_<TSP>{I} (INCR NIL (TSP) (TSP)))
(INCR_<TSP>_<TSP>{N} (INCR NIL (TSP) (TSP)))
(INCR_Y_Y{N} (INCR NIL Y Y))
(INT-TO-NAT (MOVE NIL X X))
(JUMP-N_X (MOVE-N NIL PC X))
(JUMP-NN_X (MOVE-NN NIL PC X))
(JUMP-NZ_X (MOVE-NZ NIL PC X))
(JUMP-Z_X (MOVE-Z NIL PC X))
(JUMP_* (MOVE NIL PC (PC)))
(JUMP_X{SUBR} (MOVE NIL PC X))
(LSR_<C>_X_X{N} (LSR (C) X X))
(LSR_<TSP>_<TSP>{V} (LSR NIL (TSP) (TSP)))
(MOVE-C_<TSP>_* (MOVE-C NIL (TSP) (PC 1)))
(MOVE-V_<TSP>_* (MOVE-V NIL (TSP) (PC 1)))
(MOVE-Z_<TSP>_* (MOVE-Z NIL (TSP) (PC 1)))
(MOVE-N_X_* (MOVE-N NIL X (PC 1)))
(MOVE_<TSP>_* (MOVE NIL (TSP) (PC 1)))
(MOVE_<X{A}>_<TSP> (MOVE NIL (X) (TSP)))
(MOVE_<X{S}>_<TSP> (MOVE NIL (X) (TSP)))
(MOVE_CFP_CSP (MOVE NIL CFP CSP))
(MOVE_CSP_CFP (MOVE NIL CSP CFP))
(MOVE_X_* (MOVE NIL X (PC 1)))
(MOVE_X_<X{S}> (MOVE NIL X (X)))
(MOVE_X_TSP (MOVE NIL X TSP))
(MOVE_X_X (MOVE NIL X X))
(MOVE_Y_* (MOVE NIL Y (PC 1)))
(MOVE_Y_<Y{S}> (MOVE NIL Y (Y)))
(MOVE_Y_TSP (MOVE NIL Y TSP))
(NEG_<TSP>_<TSP>{I} (NEG NIL (TSP) (TSP)))
(NOT_<TSP>_<TSP>{V} (NOT NIL (TSP) (TSP)))
(OR_<TSP>{V}_X{V} (OR NIL (TSP) X))
(OR_<TSP>{B}_X{B} (OR NIL (TSP) X))
(SUB_<C>_<TSP>{A}_X{A} (SUB (C) (TSP) X))
(SUB_<C>_<TSP>{N}_X{N} (SUB (C) (TSP) X))
(SUB_<NV>_<TSP>{I}_X{I} (SUB (N V) (TSP) X))
(SUB_<TSP>{A}_X{N} (SUB NIL (TSP) X))
(SUB_X{S}_Y{N} (SUB NIL X Y))
(SUB_<TSP>{I}_X{I} (SUB NIL (TSP) X))
(SUB_<TSP>{N}_X{N} (SUB NIL (TSP) X))
(SUB_<TSP>{S}_X{S} (SUB NIL (TSP) X))
(SUB_<Z>_X{S}_Y{S} (SUB (Z) X Y))
(SUBB_<C>_X{N}_Y{N} (SUBB (C) X Y))
(SUBB_<V>_X{I}_Y{I} (SUBB (V) X Y))
(TPOP_<C>_X (MOVE (C) X (TSP 1)))
(TPOP_<X{A}> (MOVE NIL (X) (TSP 1)))
(TPOP_<X{S}> (MOVE NIL (X) (TSP 1)))
(TPOP_PC (MOVE NIL PC (TSP 1)))
(TPOP_X (MOVE NIL X (TSP 1)))
(TPOP_Y (MOVE NIL Y (TSP 1)))
(TPOP{V}_<Z>_Y (MOVE (Z) Y (TSP 1)))
(TPOP{B}_<Z>_Y (MOVE (Z) Y (TSP 1)))
(TPOP{I}_<ZN>_Y (MOVE (Z N) Y (TSP 1)))
(TPOP{N}_<Z>_Y (MOVE (Z) Y (TSP 1)))
(TPUSH_* (MOVE NIL (-1 TSP) (PC 1)))
(TPUSH_<X{A}> (MOVE NIL (-1 TSP) (X)))
(TPUSH_<X{S}> (MOVE NIL (-1 TSP) (X)))
(TPUSH_CSP (MOVE NIL (-1 TSP) CSP))
```

```
  (TPUSH_TSP (MOVE NIL (-1 TSP) TSP))
  (TPUSH_X (MOVE NIL (-1 TSP) X))
  (XOR_<TSP>_<TSP> (XOR NIL (TSP) (TSP)))
  (XOR_<TSP>{V}_X{V} (XOR NIL (TSP) X))
  (XOR_<TSP>{B}_*{B} (XOR NIL (TSP) (PC 1)))
  (XOR_<TSP>{B}_X{B} (XOR NIL (TSP) X))
  (XOR_<Z>_<TSP>_X (XOR (Z) (TSP) X)))
```

Definition.

```
(LINK-MEM PROG-SEGMENT USR-DATA-SEGMENT SYS-DATA-SEGMENT LINK-TABLES
    WORD-SIZE)
   =
(APPEND (LINK-SEGMENT PROG-SEGMENT
                      LINK-TABLES WORD-SIZE)
        (APPEND (LINK-SEGMENT USR-DATA-SEGMENT LINK-TABLES WORD-SIZE)
                (LINK-SEGMENT SYS-DATA-SEGMENT LINK-TABLES WORD-SIZE)))
```

Definition.

```
(LINK-SEGMENT SEGMENT LINK-TABLES WORD-SIZE)
   =
(IF (NLISTP SEGMENT)
    NIL
    (APPEND (LINK-AREA (CDAR SEGMENT)
                       LINK-TABLES WORD-SIZE)
            (LINK-SEGMENT (CDR SEGMENT)
                          LINK-TABLES WORD-SIZE)))
```

Definition.

```
(LINK-TABLE-FOR-LABELS LST ADDR0)
   =
(COND ((NLISTP LST) NIL)
      ((LABELLEDP (CAR LST))
       (CONS (CONS (CADAR LST) ADDR0)
             (LINK-TABLE-FOR-LABELS (CDR LST)
                                    (ADD1 ADDR0))))
      (T (LINK-TABLE-FOR-LABELS (CDR LST)
                                (ADD1 ADDR0))))
```

Definition.

```
(LINK-TABLE-FOR-PROG-LABELS SEGMENT ADDR0)
   =
(IF (NLISTP SEGMENT)
    NIL
    (CONS (CONS (CAAR SEGMENT)
                (LINK-TABLE-FOR-LABELS (CDAR SEGMENT)
                                       ADDR0))
          (LINK-TABLE-FOR-PROG-LABELS (CDR SEGMENT)
                                      (PLUS ADDR0
                                            (LENGTH (CDAR SEGMENT))))))
```

Definition.

```
(LINK-TABLE-FOR-SEGMENT SEGMENT ADDR0)
   =
(IF (NLISTP SEGMENT)
    NIL
    (CONS (CONS (CAAR SEGMENT) ADDR0)
          (LINK-TABLE-FOR-SEGMENT (CDR SEGMENT)
                                  (PLUS ADDR0
                                        (LENGTH (CDAR SEGMENT))))))
```

Definition.
```
(LINK-WORD X LINK-TABLES WORD-SIZE)
   =
(IF (ICODE-INSTRUCTIONP X)
    (LINK-INSTR-WORD X WORD-SIZE)
    (LINK-DATA-WORD X LINK-TABLES WORD-SIZE))
```

Definition.
```
(LOAD P)
   =
(I->M (R->I (P->R P)))
```

Definition.
```
(LOCAL-VARS D)
   =
(APPEND (FORMAL-VARS D)
        (STRIP-CARS (TEMP-VAR-DCLS D)))
```

Shell Definition.
Add the shell **M-STATE** of **6** arguments, with
recognizer function symbol **M-STATEP**, and
accessors **M-REGS**, **M-C-FLG**, **M-V-FLG**, **M-N-FLG**, **M-Z-FLG**
and **M-MEM**.

Definition.
```
(MCI INS WORD-SIZE)
   =
(PACK-INSTRUCTION (EXTRACT-OP (CAR INS))
                  (EXTRACT-MOVE-BIT (CAR INS))
                  0
                  (EXTRACT-CVNZ (CADR INS))
                  (EXTRACT-MODE (CADDR INS))
                  (EXTRACT-REG (CADDR INS))
                  (EXTRACT-MODE (CADDDR INS))
                  (EXTRACT-REG (CADDDR INS))
                  WORD-SIZE)
```

Definition.
```
(NAME D)
   =
(CAR D)
```

Definition.
```
(NAT-0S N)
   =
(IF (ZEROP N)
    NIL
    (CONS (TAG 'NAT 0)
          (NAT-0S (SUB1 N))))
```

Definition.
```
(NAT-TO-BV N SIZE)
   =
(IF (ZEROP SIZE)
    (BTM)
    (BITV (IF (ZEROP (REMAINDER N 2)) F T)
          (NAT-TO-BV (QUOTIENT N 2)
                     (SUB1 SIZE))))
```

Definition.
```
(OFFSET-FROM-CSP VAR PROGRAM)
   =
(FIND-POSITION-OF-VAR VAR
                      (LOCAL-VARS PROGRAM))
```

Definition.
```
(P->R P)
   =
(R-STATE (P-PC P)
         (P->R_CFP (P-CTRL-STK P)
                   (P-MAX-CTRL-STK-SIZE P))
         (P->R_CSP (P-CTRL-STK P)
                   (P-MAX-CTRL-STK-SIZE P))
         (P->R_TSP (P-TEMP-STK P)
                   (P-MAX-TEMP-STK-SIZE P))
         '(NAT 0)
         '(NAT 0)
         '(BOOL F)
         '(BOOL F)
         '(BOOL F)
         '(BOOL F)
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P->R_SYS-DATA-SEGMENT (P-CTRL-STK P)
                                (P-MAX-CTRL-STK-SIZE P)
                                (P-TEMP-STK P)
                                (P-MAX-TEMP-STK-SIZE P))
         (P-WORD-SIZE P)
         (P-PSW P))
```

Definition.
```
(P->R_CFP STK MAX)
   =
(SUB-ADDR (P->R_CSP (POP STK) MAX) 2)
```

Definition.
```
(P->R_CSP STK MAX)
   =
(TAG 'SYS-ADDR
     (CONS 'CSTK
           (DIFFERENCE MAX
                       (P-CTRL-STK-SIZE STK))))
```

Definition.
```
(P->R_CTRL-STK STK MAX)
   =
(CONS 'CSTK
      (APPEND (NAT-0S (DIFFERENCE MAX
                                  (P-CTRL-STK-SIZE STK)))
              (APPEND (P->R_CTRL-STK1 STK MAX)
                      (LIST (TAG 'NAT 0)))))
```

Definition.
```
(P->R_CTRL-STK1 STK MAX)
   =
(IF (NLISTP STK)
    NIL
    (APPEND (P->R_P-FRAME (TOP STK) (POP STK) MAX)
            (P->R_CTRL-STK1 (POP STK) MAX)))
```

Definition.
```
(P->R_P-FRAME PFRAME STK MAX)
   =
(APPEND (STRIP-CDRS (BINDINGS PFRAME))
        (LIST (P->R_CFP STK MAX)
              (RET-PC PFRAME)))
```

Definition.
```
(P->R_SYS-DATA-SEGMENT CTRL-STK MAX-CTRL-STK-SIZE TEMP-STK
    MAX-TEMP-STK-SIZE)
   =
(LIST (P->R_CTRL-STK CTRL-STK MAX-CTRL-STK-SIZE)
      (P->R_TEMP-STK TEMP-STK MAX-TEMP-STK-SIZE)
      (LIST 'FULL-CTRL-STK-ADDR
            (TAG 'SYS-ADDR '(CSTK . 0)))
      (LIST 'FULL-TEMP-STK-ADDR
            (TAG 'SYS-ADDR '(TSTK . 0)))
      (LIST 'EMPTY-TEMP-STK-ADDR
            (TAG 'SYS-ADDR
                 (CONS 'TSTK MAX-TEMP-STK-SIZE))))
```

Definition.
```
(P->R_TEMP-STK TEMP-STK MAX-TEMP-STK-SIZE)
   =
(CONS 'TSTK
      (APPEND (NAT-0S (DIFFERENCE MAX-TEMP-STK-SIZE
                                  (LENGTH TEMP-STK)))
              (APPEND TEMP-STK
                      (LIST (TAG 'NAT 0)))))
```

Definition.
```
(P->R_TSP STK MAX)
   =
(TAG 'SYS-ADDR
     (CONS 'TSTK
           (DIFFERENCE MAX (LENGTH STK))))
```

Definition.
```
(P-CTRL-STK-SIZE CTRL-STK)
   =
(IF (NLISTP CTRL-STK)
    0
    (PLUS (P-FRAME-SIZE (TOP CTRL-STK))
          (P-CTRL-STK-SIZE (CDR CTRL-STK))))
```

Definition.
```
(P-FRAME-SIZE FRAME)
   =
(PLUS 2 (LENGTH (BINDINGS FRAME)))
```

Shell Definition.
Add the shell **P-STATE** of **9** arguments, with recognizer function symbol **P-STATEP**, and accessors **P-PC**, **P-CTRL-STK**, **P-TEMP-STK**, **P-PROG-SEGMENT**, **P-DATA-SEGMENT**, **P-MAX-CTRL-STK-SIZE**, **P-MAX-TEMP-STK-SIZE**, **P-WORD-SIZE** and **P-PSW**.

Definition.
```
(PACK-INSTRUCTION OP MOVE-BIT INT-BIT CVNZ MODE-B REG-B MODE-A REG-A
    WORD-SIZE)
   =
(NAT-TO-BV (PLUS (TIMES OP (EXP 2 16))
                 (TIMES MOVE-BIT (EXP 2 15))
                 (TIMES INT-BIT (EXP 2 14))
                 (TIMES CVNZ (EXP 2 10))
                 (TIMES MODE-B (EXP 2 8))
                 (TIMES REG-B (EXP 2 5))
                 (TIMES MODE-A (EXP 2 3))
                 REG-A)
           WORD-SIZE)
```

Definition.
```
(PC LAB PROGRAM)
   =
(TAG 'PC
     (CONS (NAME PROGRAM)
           (FIND-LABEL LAB
                       (PROGRAM-BODY PROGRAM))))
```

Definition.
```
(POP STK)
   =
(CDR STK)
```

Definition.
```
(PROG-LABEL-TABLES LINK-TABLES)
   =
(CADR LINK-TABLES)
```

Definition.
```
(PROG-LINKS LINK-TABLES)
   =
(CAR LINK-TABLES)
```

Definition.
```
(PROGRAM-BODY D)
   =
(CDDDR D)
```

Definition.
```
(R->I R)
    =
(I-STATE (R->I_PC (R-PC R) (R-PROG-SEGMENT R))
         (R-CFP R)
         (R-CSP R)
         (R-TSP R)
         (R-X R)
         (R-Y R)
         (R-C-FLG R)
         (R-V-FLG R)
         (R-N-FLG R)
         (R-Z-FLG R)
         (ICOMPILE (R-PROG-SEGMENT R))
         (R-USR-DATA-SEGMENT R)
         (R-SYS-DATA-SEGMENT R)
         (R-WORD-SIZE R)
         (R->I_PSW (R-PSW R)))
```

Definition.
```
(R->I_PC PC PROGRAMS)
    =
(TAG 'IPC
     (CONS (AREA-NAME PC)
           (FIND-LABEL (UNTAG PC)
                       (CDR (ICOMPILE-PROGRAM (DEFINITION (AREA-NAME PC)
                                                          PROGRAMS))))))
```

Definition.
```
(R->I_PSW PSW)
    =
(IF (EQUAL PSW 'HALT) 'RUN PSW)
```

Shell Definition.
Add the shell **R-STATE** of **15** arguments, with
recognizer function symbol **R-STATEP**, and
accessors **R-PC**, **R-CFP**, **R-CSP**, **R-TSP**, **R-X**, **R-Y**,
R-C-FLG, **R-V-FLG**, **R-N-FLG**, **R-Z-FLG**,
R-PROG-SEGMENT, **R-USR-DATA-SEGMENT**, **R-SYS-DATA-SEGMENT**,
R-WORD-SIZE and **R-PSW**.

Definition.
```
(RET-PC FRAME)
    =
(CADR FRAME)
```

Definition.
```
(REVERSE X)
    =
(IF (NLISTP X)
    NIL
    (APPEND (REVERSE (CDR X))
            (LIST (CAR X))))
```

Definition.
```
(SEGMENT-LENGTH SEGMENT)
   =
(IF (NLISTP SEGMENT)
    0
    (PLUS (LENGTH (CDAR SEGMENT))
          (SEGMENT-LENGTH (CDR SEGMENT))))
```

Definition.
```
(STRIP-CDRS ALIST)
   =
(IF (NLISTP ALIST)
    NIL
    (CONS (CDAR ALIST)
          (STRIP-CDRS (CDR ALIST))))
```

Definition.
```
(SUB-ADDR ADDR N)
   =
(TAG (TYPE ADDR)
     (SUB-ADP (UNTAG ADDR) N))
```

Definition.
```
(SUB-ADP ADP N)
   =
(CONS (ADP-NAME ADP)
      (DIFFERENCE (ADP-OFFSET ADP) N))
```

Definition.
```
(SUBR-TO-BV SUBR PROG-LINKS WORD-SIZE)
   =
(NAT-TO-BV (BASE-ADDRESS SUBR PROG-LINKS)
           WORD-SIZE)
```

Definition.
```
(SYS-ADDR-TO-BV ADP SYS-DATA-LINKS WORD-SIZE)
   =
(NAT-TO-BV (ABSOLUTE-ADDRESS ADP SYS-DATA-LINKS)
           WORD-SIZE)
```

Definition.
```
(SYS-DATA-LINKS LINK-TABLES)
   =
(CADDDR LINK-TABLES)
```

Definition.
```
(TAG TYPE OBJ)
   =
(LIST TYPE OBJ)
```

Definition.
```
(TC-TO-BV X SIZE)
   =
(IF (NEGATIVEP X)
    (INCR T
          (COMPL (NAT-TO-BV (NEGATIVE-GUTS X) SIZE)))
    (NAT-TO-BV X SIZE))
```

Definition.
```
(TEMP-VAR-DCLS D)
   =
(CADDR D)
```

Definition.
```
(TOP STK)
   =
(CAR STK)
```

Definition.
```
(TYPE CONST)
   =
(CAR CONST)
```

Definition.
```
(UNLABEL X)
   =
(IF (LABELLEDP X) (CADDDR X) X)
```

Definition.
```
(UNTAG CONST)
   =
(CADR CONST)
```

Definition.
```
(USR-DATA-LINKS LINK-TABLES)
   =
(CADDR LINK-TABLES)
```

Definition.
```
(V-APPEND A B)
   =
(IF (BTMP A)
    B
    (BITV (BIT A) (V-APPEND (VEC A) B)))
```

Definition.
```
(XOR X Y)
   =
(COND (X (IF Y F T)) (Y T) (T F))
```

10. The Formal Correctness Theorem

The correctness theorem for the FM8502 implementation of Piton is

Theorem. FM8502 Piton is Correct

```
(IMPLIES (AND (PROPER-P-STATEP P0)
              (P-LOADABLEP P0)
              (EQUAL (P-WORD-SIZE P0) 32)
              (EQUAL PN (P P0 N))
              (NOT (ERRORP (P-PSW PN)))
              (EQUAL TS (TYPE-SPECIFICATION
                            (P-DATA-SEGMENT PN))))
         (EQUAL (P-DATA-SEGMENT PN)
                (DISPLAY-M-DATA-SEGMENT
                         (FM8502 (LOAD P0)
                                 (FM8502-CLOCK P0 N))
                         TS
                         (LINK-TABLES P0)))).
```

We have already presented the formal definitions of **PROPER-P-STATEP**, **P**, **FM8502**, **LOAD** and the accessors for the **P-STATE** shell, including **P-WORD-SIZE**, **P-PSW**, **P-DATA-SEGMENT**. The remaining functions used in the statement of correctness are **P-LOADABLEP**, **ERRORP**, **TYPE-SPECIFICATION**, **DISPLAY-M-DATA-SEGMENT**, **LINK-TABLES**, and **FM8502-CLOCK**. All but the last are defined in this chapter.

Recall our discussion of the correctness theorem and in particular of the clock expression, **(FM8502-CLOCK P0 N)**. We regard **FM8502-CLOCK** as a ‘‘witness function’’ for what is informally understood as existential quantification on the number of steps that FM8502 must be run to duplicate a Piton computation. We do not exhibit the definition of **FM8502-CLOCK** in this document because we think of it as a definition specific to our particular proof. Suffice it to say that **FM8502-CLOCK** can be constructively defined—indeed, it *is* constructively defined in our proof. Its definition is somewhat larger than that of **P** itself and is structurally isomorphic. **(FM8502-CLOCK P0 N)** steps forward from **P0**. On each iteration **FM8502-CLOCK** uses a user-defined function specific to our implementation to determine how many FM8502 machine instructions are executed for the current Piton instruction in the current environment.[17] **FM8502-CLOCK** sums the instruction counts and iterates until either **N** steps are taken or the Piton computation halts normally or with an error. With the exception of the footnote on page 221 we do not discuss **FM8502-CLOCK** further in this document.

The rest of this chapter is an alphabetical listing of the remaining functions used in the correctness theorem. The ‘‘entry points’’ into the definitions are the functions **P-LOADABLEP**, **ERRORP**, **TYPE-SPECIFICATION**, **DISPLAY-M-DATA-SEGMENT** and **LINK-TABLES**. It is noteworthy that all of the definitions listed in the preceding Chapters 7-9, plus those of this chapter, are necessary *simply to state* the correctness result. The *proof* of the correctness result requires the definition of **FM8502-CLOCK** and of hundreds of other functions.

Reader uninterested in pursuing the formal definitions of the concepts involved in the correctness theorem should skip to page 211.

---

[17] Sometimes the number of machine code instructions varies according to which path is taken through the i-code generated.

Definition.
```
(AREA-TYPE-SPECIFICATION AREA)
   =
(CONS (CAR AREA)
      (TYPE-LST (CDR AREA)))
```

Definition.
```
(ASSOC-CDRP N ALIST)
   =
(COND ((NLISTP ALIST) F)
      ((EQUAL N (CDAR ALIST)) T)
      (T (ASSOC-CDRP N (CDR ALIST))))
```

Definition.
```
(BV-TO-ADDR BV USR-DATA-LINKS)
   =
(INVERT-ABSOLUTE-ADDRESS (BV-TO-NAT BV)
                         USR-DATA-LINKS)
```

Definition.
```
(BV-TO-BITV BV)
   =
(IF (OR (NOT (BITVP BV)) (EQUAL BV (BTM)))
    NIL
    (APPEND (BV-TO-BITV (VEC BV))
            (LIST (IF (BIT BV) 1 0))))
```

Definition.
```
(BV-TO-BOOL BV)
   =
(IF (BIT BV) 'T 'F)
```

Definition.
```
(BV-TO-LABEL BV PROG-LABEL-TABLES)
   =
(INVERT-LABEL-ADDRESS (BV-TO-NAT BV)
                      PROG-LABEL-TABLES)
```

Definition.
```
(BV-TO-SUBR BV PROG-LINKS)
   =
(INVERT-BASE-ADDRESS (BV-TO-NAT BV)
                     PROG-LINKS)
```

Definition.
```
(BV-TO-SYS-ADDR BV SYS-DATA-LINKS)
   =
(INVERT-ABSOLUTE-ADDRESS (BV-TO-NAT BV)
                         SYS-DATA-LINKS)
```

Definition.
```
(DISPLAY-ARRAY TYPE-LST M-ADDR M-MEM LINK-TABLES)
   =
(IF (NLISTP TYPE-LST)
    NIL
    (CONS (UNLINK-DATA-WORD (CAR TYPE-LST)
                            (GET M-ADDR M-MEM)
                            LINK-TABLES)
          (DISPLAY-ARRAY (CDR TYPE-LST)
                         (ADD1 M-ADDR)
                         M-MEM LINK-TABLES)))
```

Definition.
```
(DISPLAY-DATA-AREA AREA-TYPE-SPEC M-MEM LINK-TABLES)
   =
(CONS (CAR AREA-TYPE-SPEC)
      (DISPLAY-ARRAY (CDR AREA-TYPE-SPEC)
                     (BASE-ADDRESS (CAR AREA-TYPE-SPEC)
                                   (USR-DATA-LINKS LINK-TABLES))
                     M-MEM LINK-TABLES))
```

Definition.
```
(DISPLAY-DATA-SEGMENT TYPE-SPEC M-MEM LINK-TABLES)
   =
(IF (NLISTP TYPE-SPEC)
    NIL
    (CONS (DISPLAY-DATA-AREA (CAR TYPE-SPEC)
                             M-MEM LINK-TABLES)
          (DISPLAY-DATA-SEGMENT (CDR TYPE-SPEC)
                                M-MEM LINK-TABLES)))
```

Definition.
```
(DISPLAY-M-DATA-SEGMENT M TYPE-SPEC LINK-TABLES)
   =
(DISPLAY-DATA-SEGMENT TYPE-SPEC
                      (M-MEM M)
                      LINK-TABLES)
```

Definition.
```
(ERRORP PSW)
   =
(AND (NOT (EQUAL PSW 'RUN))
     (NOT (EQUAL PSW 'HALT)))
```

Definition.
```
(FIND-CONTAINING-AREA-NAME N LINK-TABLE)
   =
(COND ((NLISTP LINK-TABLE) 0)
      ((NLISTP (CDR LINK-TABLE))
       (CAAR LINK-TABLE))
      ((AND (NOT (LESSP N (CDAR LINK-TABLE)))
            (LESSP N (CDADR LINK-TABLE)))
       (CAAR LINK-TABLE))
      (T (FIND-CONTAINING-AREA-NAME N
                                    (CDR LINK-TABLE))))
```

Definition.
```
(FIND-CONTAINING-LABEL-TABLE N LABEL-TABLES)
   =
(COND ((NLISTP LABEL-TABLES) F)
      ((ASSOC-CDRP N (CDAR LABEL-TABLES))
       (CDAR LABEL-TABLES))
      (T (FIND-CONTAINING-LABEL-TABLE N
                                      (CDR LABEL-TABLES))))
```

Definition.
```
(INVERT-ABSOLUTE-ADDRESS N LINK-TABLE)
   =
(CONS (FIND-CONTAINING-AREA-NAME N
                                 LINK-TABLE)
      (DIFFERENCE N
                  (BASE-ADDRESS (FIND-CONTAINING-AREA-NAME N LINK-TABLE)
                                LINK-TABLE)))
```

Definition.
```
(INVERT-BASE-ADDRESS N LINK-TABLE)
   =
(FIND-CONTAINING-AREA-NAME N
                           LINK-TABLE)
```

Definition.
```
(INVERT-LABEL-ADDRESS N PROG-LABEL-TABLES)
   =
(INVERT-BASE-ADDRESS N
                     (FIND-CONTAINING-LABEL-TABLE N PROG-LABEL-TABLES))
```

Definition.
```
(LINK-TABLES P)
   =
(I-LINK-TABLES (R->I (P->R P)))
```

Definition.
```
(P-LOADABLEP P)
   =
(LESSP (TOTAL-P-SYSTEM-SIZE P)
       (EXP 2 (P-WORD-SIZE P)))
```

Definition.
```
(TOTAL-P-SYSTEM-SIZE P)
   =
(PLUS (SEGMENT-LENGTH (ICOMPILE (P-PROG-SEGMENT P)))
      (SEGMENT-LENGTH (P-DATA-SEGMENT P))
      (ADD1 (P-MAX-CTRL-STK-SIZE P))
      (ADD1 (P-MAX-TEMP-STK-SIZE P))
      3)
```

Definition.
```
(TYPE-LST LST)
   =
(IF (NLISTP LST)
    NIL
    (CONS (TYPE (CAR LST))
          (TYPE-LST (CDR LST))))
```

Definition.

```
(TYPE-SPECIFICATION SEGMENT)
   =
(IF (NLISTP SEGMENT)
    NIL
    (CONS (AREA-TYPE-SPECIFICATION (CAR SEGMENT))
          (TYPE-SPECIFICATION (CDR SEGMENT))))
```

Definition.

```
(UNLINK-DATA-WORD TYPE BV LINK-TABLES)
   =
(CASE TYPE
      (NAT       (TAG 'NAT (BV-TO-NAT BV)))
      (INT       (TAG 'INT (BV-TO-TC BV)))
      (BITV      (TAG 'BITV (BV-TO-BITV BV)))
      (BOOL      (TAG 'BOOL (BV-TO-BOOL BV)))
      (ADDR      (TAG 'ADDR
                      (BV-TO-ADDR BV
                                  (USR-DATA-LINKS LINK-TABLES))))
      (SUBR      (TAG 'SUBR
                      (BV-TO-SUBR BV
                                  (PROG-LINKS LINK-TABLES))))
      (SYS-ADDR  (TAG 'SYS-ADDR
                      (BV-TO-SYS-ADDR BV
                                      (SYS-DATA-LINKS LINK-TABLES))))
      (PC        (TAG 'PC
                      (BV-TO-LABEL BV
                                   (PROG-LABEL-TABLES LINK-TABLES))))
      (OTHERWISE '(UNRECOGNIZED I-LEVEL TYPE)))
```

11. Proof of the Correctness Theorem

The FM8502 implementation of Piton is embodied in the function **LOAD**, which is defined as the composition of three functions,

Definition.
```
(LOAD P)
   =
(I->M (R->I (P->R P))),
```

one for each of the phases: resource representation (**P->R**), compilation (**R->I**) and link-assembling (**I->M**).

The key to our proof of the correctness of the implementation is to prove the correctness of each of the three phases separately. We then combine these three lemmas to get our main correctness proof.

But what does it mean to say that the resource representation phase is correct? What does it mean to say that the compiler is correct, in isolation from the link-assembler and FM8502? To formalize the correctness of these phases independently we must formally specify two abstract machines intermediate between Piton and FM8502. We call the first of these machines the ‘‘**R**’’ machine, which interprets r-states, and the second the ‘‘**I**’’ machine, which interprets i-states. In addition, we define a third machine, called the ‘‘**M**’’ machine, which is just FM8502, but defined in the same style as our other machines rather than via **SOFT**.

The organization of this description of our proof is as follows. First we describe **R**, **I**, and **M**. Then we state the fundamental properties relating the various machines via the functions **P->R**, **R->I** and **I->M** and sketch the proofs of each relation. Finally, we use these fundamental properties (along with several other properties) to prove the main correctness result.

11.1. The R Machine

The **R** machine is very similar to the **P** machine in that its programming language is Piton. However, its resources (namely the stacks) are represented in terms of the system data segment and the registers. For example, where the **P** machine implements a push by **CONS**ing, the **R** machine does it via decrementing the stack pointer and depositing into the indicated position of the stack array.

Let us consider the Piton **PUSH-CONSTANT** instruction. The **P** machine specification for this instruction is that it increments the program counter and pushes the unabbreviated operand onto the temporary stack. Formally this is rendered

Definition.

```
(P-PUSH-CONSTANT-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (UNABBREVIATE-CONSTANT (CADR INS) P)
               (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN).
```

The function **PUSH** here is just **CONS**; the temporary stack is a list and the new object is consed onto the front of that list.

The **R** machine specification of **PUSH-CONSTANT** is

Definition.

```
(R-PUSH-CONSTANT-STEP INS R)
   =
(R-STATE (ADD1-R-PC R)
         (R-CFP R)
         (R-CSP R)
         (PUSH-STK (R-TSP R))
         (R-X R)
         (R-Y R)
         (R-C-FLG R)
         (R-V-FLG R)
         (R-N-FLG R)
         (R-Z-FLG R)
         (R-PROG-SEGMENT R)
         (R-USR-DATA-SEGMENT R)
         (DEPOSIT (IF (EQUAL (CADR INS) 'PC)
                      (ADD1-R-PC R)
                      (IF (NLISTP (CADR INS))
                          (PC (CADR INS)
                              (R-CURRENT-PROGRAM R))
                          (CADR INS)))
                  (PUSH-STK (R-TSP R))
                  (R-SYS-DATA-SEGMENT R))
         (R-WORD-SIZE R)
         'RUN).
```

The function **PUSH-STK** used above decrements a tagged address pair by one, e.g., **(PUSH-STK '(SYS-ADDR (TSTK . 25)))** is equal to **'(SYS-ADDR (TSTK . 24))**. **(DEPOSIT obj addr segment)** writes **obj** into address **addr** of **segment**, where **addr** is a tagged address pair (specifying a name and an offset) and **segment** is an association list pairing area names with arrays.

Observe that the **R** machine specification of **PUSH-CONSTANT** increments the program counter, decrements the **R-TSP** register, and writes the unabbreviated operand to the system data segment location addressed by the decremented **R-TSP** register. That is, the **R** machine implements the Piton instructions using the resources of FM8502.

The situation is a little more subtle than this example might suggest. Consider the Piton **PUSH-LOCAL** instruction. The **P** machine specification is

Definition.

```
(P-PUSH-LOCAL-STEP INS P)
   =
(P-STATE (ADD1-P-PC P)
         (P-CTRL-STK P)
         (PUSH (LOCAL-VAR-VALUE (CADR INS) (P-CTRL-STK P))
               (P-TEMP-STK P))
         (P-PROG-SEGMENT P)
         (P-DATA-SEGMENT P)
         (P-MAX-CTRL-STK-SIZE P)
         (P-MAX-TEMP-STK-SIZE P)
         (P-WORD-SIZE P)
         'RUN).
```

The **R** machine specification is

Definition.

```
(R-PUSH-LOCAL-STEP INS R)
   =
(R-STATE (ADD1-R-PC R)                                                    ; (a)
         (R-CFP R)
         (R-CSP R)
         (PUSH-STK (R-TSP R))                                             ; (b)
         (ADD-ADDR (R-CSP R)                                              ; (c)
                   (OFFSET-FROM-CSP
                    (CADR INS)
                    (R-CURRENT-PROGRAM R)))
         (R-Y R)
         (R-C-FLG R)
         (R-V-FLG R)
         (R-N-FLG R)
         (R-Z-FLG R)
         (R-PROG-SEGMENT R)
         (R-USR-DATA-SEGMENT R)
         (DEPOSIT (FETCH (ADD-ADDR (R-CSP R)                              ; (d)
                                   (OFFSET-FROM-CSP
                                    (CADR INS)
                                    (R-CURRENT-PROGRAM R)))
                         (R-SYS-DATA-SEGMENT R))
                  (PUSH-STK (R-TSP R))
                  (R-SYS-DATA-SEGMENT R))
         (R-WORD-SIZE R)
         'RUN).
```

The **ADD-ADDR** expression, which occurs twice above (in lines (c) and (d)), computes the address at which the value of the indicated local variable is stored. The address is computed by adding to the current control stack pointer—the contents of the **R-CSP** register—the offset of the local variable among the current program's local variables. Call that address **addr**.

An informal reading of the **R** machine specification for **PUSH-LOCAL** is that it (a) increments the program counter, (b) decrements the **R-TSP** register, (c) sets the **R-X** register to **addr** and (d) deposits into the system data area at the new top of the temporary stack the contents of **addr**. Steps (a), (b), and (d) are intuitively necessary and sufficient.

Why, however, does the **R** machine set the **R-X** register? It turns out that no **R** machine instruction inspects the values of the temporary registers, **R-X**, **R-Y** and the four flags. However, many of the instructions set those registers and flags. Why? The answer is that the **R** machine does more than just implement the Piton instructions with the FM8502 resources. It implements the Piton instructions with the FM8502 resources *in exactly the same way our compiler does*.

For example, the compiled code for **PUSH-LOCAL** is generated by

Definition.
```
(ICODE-PUSH-LOCAL INS PCN PROGRAM)
   =
(LIST '(MOVE_X_*)
      (TAG 'NAT (OFFSET-FROM-CSP (CADR INS) PROGRAM))
      '(ADD_X{N}_CSP)
      '(TPUSH_<X{S}>)).
```

Observe that this code moves into the **x** register the offset of the required local, then adds the **csp** register to that, leaving the result in **x**, and then pushes the contents of the address in **x** onto the temporary stack. Thus, the implementation of **PUSH-LOCAL** has the additional side-effect on FM8502 of setting the **x** register to the address of the local pushed. The **R** machine specification of **PUSH-LOCAL** faithfully describes this implementation of **PUSH-LOCAL**.

Formally, **(R R N)** is defined to step forward from the r-state **R N** times, using the function **R-STEP**. **R-STEP** checks the precondition of the current instruction and, if it is satisfied, produces the next state with the step function for the current instruction.

The **R** machine can thus be imagined by starting with the **P** machine and modifying every precondition and step function in two ways. First, replace all references to the abstract stacks by fetches and deposits on the registers and system data segment so as to mimic at the **R**-level what the **P** machine does. Second, add the six temporary registers and flags so that the **R** machine correctly describes the final values of those FM8502 resources at the conclusion of the i-code generated for each instruction.

11.2. The I Machine

The **I** machine operates on i-states. Recall that i-states are like r-states except that the programs are written in i-code, not Piton. Thus, the **I** machine interprets i-code using the same stack representation as the **R** machine. The **I** machine is defined in a style very similar to our other interpreters. **(I I N)** steps the i-state **I** forward **N** times, using the precondition and step functions specific to the current instruction. We illustrate the **I** machine by simply exhibiting the step functions for four of the 87 i-code instructions.

Here is the step function for the i-code instruction **MOVE_X_***, which moves the next word of the instruction stream into register **x** and increments the program counter by two:

Definition .

```
(I-MOVE_X_*-STEP I)
    =
(I-STATE (ADD2-I-PC I)
         (I-CFP I)
         (I-CSP I)
         (I-TSP I)
         (I-NEXTWORD I)
         (I-Y I)
         (I-C-FLG I)
         (I-V-FLG I)
         (I-N-FLG I)
         (I-Z-FLG I)
         (I-PROG-SEGMENT I)
         (I-USR-DATA-SEGMENT I)
         (I-SYS-DATA-SEGMENT I)
         (I-WORD-SIZE I)
         'RUN).
```

The function **I-NEXTWORD**, used above, fetches the contents of the program segment address one greater than the current program counter,

Definition .

```
(I-NEXTWORD I)
    =
(UNLABEL (FETCH (ADD1-I-PC I) (I-PROG-SEGMENT I))).
```

Next we exhibit the step function for **TPUSH_<X{S}>**, which pushes onto the temporary stack the contents of the system address contained in the **X** register.

Definition .

```
(I-TPUSH_<X{S}>-STEP I)
    =
(I-STATE (ADD1-I-PC I)
         (I-CFP I)
         (I-CSP I)
         (PUSH-STK (I-TSP I))
         (I-X I)
         (I-Y I)
         (I-C-FLG I)
         (I-V-FLG I)
         (I-N-FLG I)
         (I-Z-FLG I)
         (I-PROG-SEGMENT I)
         (I-USR-DATA-SEGMENT I)
         (DEPOSIT (FETCH (I-X I)
                         (I-SYS-DATA-SEGMENT I))
                  (PUSH-STK (I-TSP I))
                  (I-SYS-DATA-SEGMENT I))
         (I-WORD-SIZE I)
         'RUN)
```

Here is the instruction **ADD_<TSP>{N}_X{N}**, which replaces the top of the temporary stack by the natural number sum of the current top and the contents of the **X** register, both of which must be naturals. The precondition for the instruction checks that both operands of the addition are tagged naturals and that their sum is representable.

Definition.

```
(I-ADD_<TSP>{N}_X{N}-STEP I)
   =
(I-STATE (ADD1-I-PC I)
         (I-CFP I)
         (I-CSP I)
         (I-TSP I)
         (I-X I)
         (I-Y I)
         (I-C-FLG I)
         (I-V-FLG I)
         (I-N-FLG I)
         (I-Z-FLG I)
         (I-PROG-SEGMENT I)
         (I-USR-DATA-SEGMENT I)
         (DEPOSIT
          (TAG 'NAT
               (PLUS (UNTAG (FETCH (I-TSP I)
                                   (I-SYS-DATA-SEGMENT I)))
                     (UNTAG (I-X I))))
          (I-TSP I)
          (I-SYS-DATA-SEGMENT I))
         (I-WORD-SIZE I)
         'RUN)
```

Recall that i-code instructions are mapped into machine code via the table shown on page 82. Some distinct i-code instructions are mapped to the same assembly instruction (and hence to the same machine instruction). For example, both **ADD_<TSP>{N}_X{N}**, shown above, and its integer counterpart, **ADD_<TSP>{I}_X{I}**, are mapped to **(ADD () (TSP) X)**. The **I** machine differentiates these two i-code instructions. In particular, the **I** machine semantics for the integer version of the instruction uses integer addition, **IPLUS**, and tags the result with **INT** rather than **NAT**. Similarly, the previously discussed pair, **XOR_<TSP>{V}_X{V}** and **XOR_<TSP>{B}_X{B}** (page 83), both of which are mapped to **(XOR () (TSP) X)**, are distinct at the **I** level. The first computes the componentwise exclusive-or of two bit vectors. The second computes the exclusive-or of two Booleans.

The main point is that the **I** machine provides abstract data types, even though its instruction set is in 1:1 correspondence with FM8502. The compiler will be proved correct with respect to the **I** machine. It is not until we drop down through the linker that addition on the naturals becomes the same operation as addition on the integers (through the wonders of twos complement representation). Similarly, after dropping through the linker, componentwise exclusive-or becomes the same operation as Boolean exclusive-or (because the Booleans **T** and **F** will be represented as bit vectors). Thus, the ‘‘mnemonic device’’ of annotating otherwise identical i-code instructions with data type information is really fundamental to our strategy of separating the compiler proof from the link-assembly proof.

This is an important point that bears repeating because it impacts the way compilers should be written. Suppose the compiler had been defined so as to generate assembly language directly rather than the annotated assembly language of i-code. For example, suppose the compiler generated **(ADD () (TSP) X)** both where it now generates **ADD_<TSP>{N}_X{N}** and where it generates **ADD_<TSP>{I}_X{I}**. This is certainly adequate if we simply intend to link-assemble the output of the compiler and run it. But if we intend to prove the compiler correct we must either (a) prove it in conjunction with the link-assembler or (b) give semantics to the assembly code independent of the link-assembler. Adopting strategy (a) vastly complicates the compiler proof. The complication does not come solely from the introduction of the bit-vector representation of integers and naturals but also from the bit-vector representation of data and program addresses. These addresses are readily related to the (now intermediate) assembly code but difficult to relate to the Piton source code. Adopting strategy (b), namely, giving semantics to the assembly code independent of the linker is not always possible.

In our example of addition strategy (b) works: define the **ADD** instruction to do a tagged integer addition if the operands are both tagged integers and a tagged natural addition otherwise (the preconditions on the Piton instructions can be used to insure that **ADD** is only executed when both operands are of the same numeric type). But there is not always enough information in the operands (or the state) to decide which of two high-level operations is to be performed. For example, both the i-code instruction **MOVE_X_X** and the i-code instruction **INT-TO-NAT** map to **(MOVE NIL X X)**. The former instruction is a no-op. The latter instruction changes the top of the temporary stack from from a tagged **INT** to a tagged **NAT**. It is impossible to give a semantics to **(MOVE NIL X X)** so that it does the appropriate thing when a tagged **INT** is on top of the stack. Thus, our device of generating annotations seems necessary to factor the correctness proof.

To reinforce the idea that the **I** machine deals with abstract types, here is the instruction **ADD_X{N}_CSP**, which adds the system address in **csp** to the natural number in **x** and stores the resulting system address in **x**.

Definition.

```
(I-ADD_X{N}_CSP-STEP I)
   =
(I-STATE (ADD1-I-PC I)
         (I-CFP I)
         (I-CSP I)
         (I-TSP I)
         (ADD-ADDR (I-CSP I)
                   (UNTAG (I-X I)))
         (I-Y I)
         (I-C-FLG I)
         (I-V-FLG I)
         (I-N-FLG I)
         (I-Z-FLG I)
         (I-PROG-SEGMENT I)
         (I-USR-DATA-SEGMENT I)
         (I-SYS-DATA-SEGMENT I)
         (I-WORD-SIZE I)
         'RUN)
```

The function **ADD-ADDR** used above is defined so that, for example, **(ADD-ADDR '(SYS-ADDR (CSTK . 25)) 7)** is **'(SYS-ADDR (CSTK . 32))**.

In some ways the **I** machine is similar to FM8502. The main similarity comes from the fact that each i-code instruction translates into a single FM8502 instruction. Unlike FM8502, i-code instructions are symbolically represented (rather than encoded as bit vectors) and all data, addresses, and program counters are tagged and represented symbolically (rather than as bit vectors). Unlike FM8502, i-code instructions check for violations of type and resource preconditions and produce erroneous states when violations are found. This insures that program space is not overwritten and that no operation is done that exposes the concrete representation of the data objects.

11.3. The M Machine

Recall that **FM8502** was defined in terms of the function **SOFT** which was proved to correspond to a gate graph.

Definition.
```
(FM8502 STATE N)
   =
(FM8502->M (SOFT (M-REGS STATE)
                 (M-MEM STATE)
                 (M-C-FLG STATE)
                 (M-V-FLG STATE)
                 (M-Z-FLG STATE)
                 (M-N-FLG STATE)
                 (FM8502-ORACLE N)))
```

However, our other machines are all defined as iterated single step functions. We therefore recast the definition of **FM8502** as an iterated single stepper. The new function is called **M**.

Definition.
```
(M STATE N)
   =
(IF (ZEROP N)
    STATE
    (M (M-STEP STATE) (SUB1 N)))
```

We define **M-STEP** in such a way that if **STATE** is an m-state then **(M STATE N)** is equal to **(FM8502 STATE N)**. The only motivation behind this definition is so that during certain proofs we can conveniently talk about single-stepping the lowest level machine.

Observe that on the **M** machine, instructions and data objects are indistinguishable, the memory is not structured into segments and areas, and the read/write/execute access to the memory is uniform.

11.4. The One-Way Correspondence Lemmas

We now have five machines, **P**, **R**, **I**, **M** and **FM8502**. The last pair of machines are easily related.

Theorem. **FM8502-EQUAL-M**
```
(IMPLIES (M-STATEP M)
         (EQUAL (FM8502 M N)
                (M M N)))
```

What are the relations among the other adjacent pairs of machines?

11.4.1. P->R

Our description of the **R** machine makes it intuitively clear that it is, in some informal sense, equivalent to the **P** machine, under a certain mapping between the two different ways the stacks are represented. That mapping, or at least the half of it that goes from the abstract representation used by **P** to the more concrete representation used by **R**, is in fact just **P->R**.

Consider two alternative r-states. The first, which we shall call r_1, is obtained by running the **P** machine on some p-state **P** and then mapping down with **P->R**. The second, which we shall call r_2, is obtained by mapping **P** down with **P->R** and then running the **R** machine.

```
                 n P-steps

         P ------------------> *
         |                     |
         |                     |
    P->R |                     | P->R
         |                     |
         |                     * r1
         *-------------------> * r2

                 n R-steps
```

What is the relation between r_1 and r_2?

They are not always equal. Consider the **X** register. In r_1 the **X** register is set to the natural number 0 because that is how **P->R** initializes it. But in r_2, the value of the **X** register is determined by the last instruction executed which set that temporary register.

Are r_1 and r_2 equal if we ignore the six temporary registers and flags? No. Consider the temporary stack area, in particular, that region of the area beyond the current **tsp** register—the ‘‘inactive’’ region ‘‘above’’ the top-of-stack. In r_1 that region consists entirely of 0’s, because that is how **P->R** initializes it. In r_2 that region contains whatever data was put there by the instructions executed by the **R** machine. How can data be put above the top of the stack? On the **R** machine that is possible, by pushing the data onto the temporary stack and then popping the stack.

Are r_1 and r_2 equal if we ignore the six temporary registers and flags and the two stack regions beyond their respective stack pointers? Yes. We define the predicate **R-EQUAL** to be this sense of weak equality. We say that the **X** and **Y** registers, the condition code registers, and the inactive stack regions of an r-state are *invisible resources* of the **R** machine. We say that the rest of the r-state is *visible*. Two r-states are **R-EQUAL** iff their visible resources are identical.

The Piton machine can be related to the **R** machine via **P->R** as follows:

```
Theorem.  ONE-WAY-CORRESPONDENCE-P-R
(IMPLIES (AND (PROPER-P-STATEP P)
              (P-LOADABLEP P)
              (NOT (ERRORP (P-PSW (P P N)))))
         (R-EQUAL (P->R (P P N))
                  (R (P->R P) N))).
```

This theorem may be read as follows. Suppose **P** is a proper p-state that is loadable. Let r_1 and r_2 be as above and suppose r_1 is non-erroneous. Then r_1 is **R-EQUAL** to r_2.

We call this a ‘‘one-way correspondence’’ theorem because it captures the ‘‘equality’’ of the **P** and **R** machines while mapping states only in one direction (in this case, from the **P** level down to the **R** level).[18]

The proof of the one-way correspondence theorem for **P** and **R** is by induction on **N**. The induction step requires two key lemmas. The first is that the one-way correspondence holds for the single steppers of the two machines,

---

[18] We considered alternative formulations in which we mapped r_2 up to a p-state, with the inverse of **P->R** but rejected it because it involved the definition of many otherwise unused concepts.

Theorem. **ONE-WAY-CORRESPONDENCE-P-R-STEP**
```
(IMPLIES (AND (PROPER-P-STATEP P)
              (NOT (ERRORP (P-PSW (P-STEP P)))))
         (R-EQUAL (P->R (P-STEP P))
                  (R-STEP (P->R P)))).
```

The second is that **R-EQUAL** is a congruence relation for **R**,

Theorem. **R-EQUAL-CONGRUENCE**
```
(IMPLIES (AND (PROPER-R-STATEP R1)
              (PROPER-R-STATEP R2)
              (R-EQUAL R2 R1))
         (R-EQUAL (R R2 N)
                  (R R1 N))),
```

i.e., if two proper r-states are **R-EQUAL** then so are the states produced by running each with **R**. We have not discussed proper r-states before. Suffice it to say that they are to r-states what loadable proper p-states are to p-states. That is, the proper r-states are those loadable r-states in which all the components are well-formed and compatible, where all the checks are made on the **R**-level representation of the stacks instead of the **P**-level.

Other lemmas used in the proof of the one-way correspondence theorem include

- If a p-state is proper and can be stepped non-erroneously then the result is proper.
- If a proper p-state is loadable and can be stepped non-erroneously then the result is loadable.
- If a p-state is erroneous then running it with the **P** machine is erroneous.
- If a p-state is proper its image under **P->R** is proper (at the **R**-level).
- If an r-state is proper (at the **R**-level) then stepping it is proper (at the **R**-level).
- **R-EQUAL** is reflexive and transitive.

We leave the proof of the one-way correspondence theorem to the reader. We urge the reader to construct it.

We briefly discuss the proof of the two key lemmas, **ONE-WAY-CORRESPONDENCE-P-R-STEP** and **R-EQUAL-CONGRUENCE**.

The proof of the first lemma, **ONE-WAY-CORRESPONDENCE-P-R-STEP**, is by case analysis on the current instruction of **P**. For each of the Piton instructions we prove the corresponding one-way correspondence lemma. Here, for example, is the one-way correspondence lemma for the **PUSH-CONSTANT** instruction.

Theorem. **PUSH-CONSTANT-ONE-WAY-CORRESPONDENCE-P-R**
```
(IMPLIES (AND (EQUAL (P-PSW P) 'RUN)
              (EQUAL (CAR (P-CURRENT-INSTRUCTION P))
                     'PUSH-CONSTANT)
              (PROPER-P-STATEP P)
              (P-PUSH-CONSTANT-OKP (P-CURRENT-INSTRUCTION P)
                                   P))
         (R-EQUAL (P->R (P-PUSH-CONSTANT-STEP (P-CURRENT-INSTRUCTION P)
                                              P))
                  (R-PUSH-CONSTANT-STEP (P-CURRENT-INSTRUCTION P)
                                        (P->R P))))
```

These instruction-level lemmas are really the heart of the proof of the one-way correspondence theorem. To prove these lemmas we had to prove the basic facts relating the abstract representation of stacks to the concrete one. For example, we had to prove such facts as

- Pushing an object onto a stack and then mapping that stack down to the **R**-level produces the same thing as mapping the original stack down to the **R**-level and then decrementing the stack pointer address and writing the object at the indicated location.
- Popping an object from a stack and then mapping that stack down produces the same thing as mapping the original stack down and then incrementing the stack pointer.
- Building a new frame as specified by the **CALL** instruction, pushing it onto the control stack and then mapping that stack down produces the same thing as mapping the stack down and then pushing a certain sequence of objects.
- Finding the value of a local variable by looking in the bindings field of the top-frame of the control stack produces the same object as mapping the control stack down to the **R**-level and fetching the object at a certain offset from **csp**.

Of course, the first three ‘‘facts’’ are not actually valid unless by the phrase ‘‘produces the same thing as’’ we mean ‘‘produces the same active stack region as.’’

The proof of the second key lemma, **R-EQUAL-CONGRUENCE**, was broken down first into the analogous fact that **R-EQUAL** is a congruence relation for **R-STEP** and then into a case for each Piton instruction.

It is interesting to note that for the **P->R** proofs it is not important what the **R**-level machine does with the invisible resources. It is only important that the values of the invisible resources never affect the values of the visible ones.

11.4.2. R->I

We now move down one step and consider the relation between the **R** machine and the **I** machine. This can be formalized with **R->I**. Recall that the difference between r-states and i-states is that the former contain Piton programs and the latter contain i-code programs. **R->I** is the compiler. The appropriate lemma is

Theorem. ONE-WAY-CORRESPONDENCE-R-I
```
(IMPLIES (AND (PROPER-R-STATEP R)
              (NOT (ERRORP (R-PSW (R R N)))))
         (EQUAL (R->I (R R N))
                (I (R->I R)
                   (CLOCK R N)))).
```

This theorem may be read as follows: Suppose **R** is a proper r-state and that the result of running **R N** steps is non-erroneous. Then consider two alternative i-states. The first is obtained by running **R N** steps and mapping down with **R->I**. The second is obtained by mapping **R** down to an i-state and then running it **(CLOCK R N)** steps forward. Then these two i-states are actually equal.

The **CLOCK** function is just a modification of the **R** machine that counts the number of i-code instructions executed on behalf of each Piton instruction.[19]

The one-way correspondence result for **R->I** is beautiful because it is a strict equality, not a weak

[19] The previously discussed expression **(FM8502-CLOCK P N)** is in fact defined to be **(CLOCK (P->R P) N)**. The remarks made earlier suggesting that **FM8502-CLOCK** was derived from **P** were true in the sense that **CLOCK** is derived from **R** and **R** is derived from **P**. We could not say this earlier because the **R** machine had not been introduced.

equality as in the **P->R** case. This is so because the **R** machine accurately accounts for the use of all the resources of FM8502—even the invisible ones. However, the proof is complicated because the **I** machine has to take many steps for each single step of the **R** machine.

The proof is by induction on **N** again and the key lemma is the one-way correspondence theorem for **R-STEP**,

Theorem. **ONE-WAY-CORRESPONDENCE-R-I-STEP**
```
(IMPLIES (AND (PROPER-R-STATEP R)
              (NOT (ERRORP (R-PSW (R-STEP R)))))
         (EQUAL (R->I (R-STEP R))
                (I (R->I R)
                   (R-STEP-CLOCK R)))).
```

Note that on the left-hand side of the conclusion we have a single **R-STEP**, while on the right-hand side we run **I** a certain number of steps, namely **(R-STEP-CLOCK R)**.

To use this **R-STEP** lemma to prove the one-way correspondence theorem we need the observation

Theorem. **I-SEQUENTIAL-EXECUTION**
```
(EQUAL (I I (PLUS X Y))
       (I (I I X) Y)).
```

Intuitively, this lemma lets us piece together many short runs of **I**—each over the block of compiled code corresponding to one Piton instruction—into a single long run of **I**. This lemma is easy to prove and we do not discuss it further.

We turn instead to the proof of the one-way correspondence theorem for **R-STEP**. That theorem is the heart of the compiler proof. Once again, we split the theorem into a separate case for each Piton instruction. Here is the case for **PUSH-LOCAL**.

Theorem. **PUSH-LOCAL-ONE-WAY-CORRESPONDENCE-R-I**
```
(IMPLIES (AND (EQUAL (R-PSW R) 'RUN)
              (EQUAL (CAR (R-CURRENT-INSTRUCTION R))
                     'PUSH-LOCAL)
              (PROPER-R-STATEP R)
              (R-PUSH-LOCAL-OKP (R-CURRENT-INSTRUCTION R)
                                R))
         (EQUAL (R->I (R-PUSH-LOCAL-STEP (R-CURRENT-INSTRUCTION R)
                                         R))
                (I (R->I R)
                   (R-PUSH-LOCAL-STEP-CLOCK
                    (R-CURRENT-INSTRUCTION R)
                    R))))
```

Observe that the hypotheses include the assumption that the opcode of the current instruction at the **R**-level is **PUSH-LOCAL**. The conclusion involves running the **I** machine for a certain number of clock ticks on the i-state **(R->I R)**. The function **R-PUSH-LOCAL-STEP-CLOCK** determines the number of i-code instructions executed for this particular **PUSH-LOCAL** execution. In the case of **PUSH-LOCAL** the number of i-code instructions is 3 (since there are no branches in the i-code generated for **PUSH-LOCAL** the number of clock ticks for that instruction is constant). Thus, the right-hand side of the conclusion of the above theorem is equivalent to **(I (R->I R) 3)**.

The proof of the theorem requires the ability to do two kinds of reasoning. First, given a particular Piton instruction as the current instruction at the **R**-level, we must be able to deduce which i-code instructions will be executed when we run the **I**-level image of the r-state. Second, given the current **I**-level instruction, we must be able to symbolically step an i-state. The second kind of reasoning is not hard—it is just expanding the definition of **I**—provided we can deduce what the current **I**-level instruction of an i-state is. The initial i-state in which we are interested is always the image of an r-state. But after we have executed the first i-code instruction in that state, we generally have an i-state that is not the image of any r-state. We must be able to deduce what the next i-code instruction is from the fact that (a) we started with the image of an r-state, (b) we knew the opcode of the current instruction of that r-state, and (c) we are executing only as many instructions as generated for that opcode.

For example, from the assumption that the current instruction at the **R** level is a **PUSH-LOCAL** we must be able to show that the next three i-code instructions to be executed will be **(MOVE_X_*)**, **(ADD_X{N}_CSP)**, and **(TPUSH_<X{S}>)**.

The generalized version of this is that in the image of an r-state the program counter points to the beginning of the block of i-code instructions generated for the current **R**-level instruction. We formalize this as

Theorem. **FETCH-ADP-ADD-ADP-UNTAG-R->I**
```
(IMPLIES
 (AND (PROPER-R-STATEP R)
      (LESSP N
             (LENGTH
              (ICODE (GET (OFFSET (R-PC R))
                          (PROGRAM-BODY
                           (DEFINITION (ADP-NAME (UNTAG (R-PC R)))
                                       (R-PROG-SEGMENT R))))
                     (OFFSET (R-PC R))
                     (DEFINITION (ADP-NAME (UNTAG (R-PC R)))
                                 (R-PROG-SEGMENT R))))))
 (EQUAL (FETCH-ADP (ADD-ADP (UNTAG (R->I_PC (R-PC R)
                                            (R-PROG-SEGMENT R)))
                            N)
                   (ICOMPILE (R-PROG-SEGMENT R)))
        (GET N
             (ICODE (GET (OFFSET (R-PC R))
                         (PROGRAM-BODY
                          (DEFINITION (ADP-NAME (UNTAG (R-PC R)))
                                      (R-PROG-SEGMENT R))))
                    (OFFSET (R-PC R))
                    (DEFINITION (ADP-NAME (UNTAG (R-PC R)))
                                (R-PROG-SEGMENT R))))))).
```

This theorem, while apparently complicated, is really very beautiful. First, observe that the **ICODE**-expression above occurs twice. We will abbreviate that expression by **icode**. It can be informally read as ‘‘the i-code generated for the current instruction of the r-state **R**.’’ Second, the expression **(UNTAG (R->I_PC ...))** will be abbreviated **i-pc** and can be informally read as ‘‘the address pair pointing to the current instruction in the image of **R** under **R->I**.’’ Third, the **ICOMPILE**-expression can be read as ‘‘the program segment of the image of **R** under **R->I**’’ and will be abbreviated **i-prog-segment**. Making these abbreviations, the theorem becomes

Theorem. **FETCH-ADP-ADD-ADP-UNTAG-R->I**

```
(IMPLIES
 (AND (PROPER-R-STATEP R)
      (LESSP N (LENGTH icode)))
 (EQUAL (FETCH-ADP (ADD-ADP i-pc N)
                   i-prog-segment)
        (GET N icode))).
```

An informal reading of this is now easy: Let **R** be a proper r-state. Suppose **N** is a natural number less than the length of the i-code generated for the current instruction, **ins**, of **R**. Let **i** be the image of **R** under **R->I**. Increment the program counter of **i** by **N** and fetch from the program segment of **i**. What do you get? The **N**th instruction in the i-code generated for **ins**.

This theorem lets us determine the i-code instruction to be executed in any i-state produced by stepping forward from the image of an r-state, provided one has not stepped so far forward that the program counter has been pushed beyond the block of code generated for the current **R**-level instruction. The theorem is the key to our **R->I** level proofs.

Our proof also relies on such lemmas as

- The fundamental properties of fetch and deposit, e.g., that **(FETCH A1 (DEPOSIT VAL A2 SEGMENT))** is **VAL** if **A1** is **A2** and is **(FETCH A1 SEGMENT)** otherwise. These lemmas enable the i-code to create a complicated state by performing several simpler transformations sequentially.
- If x is a legal **R**-level object in some r-state then it is also a legal **I**-level object in the image of that r-state. This theorem let us establish the **I**-level preconditions from the **R**-level preconditions.

In addition, there were many facts that were needed to handle the compiled code for specific instructions.

For example, the i-code for the **TEST-INT-AND-JUMP** instruction in the case where the test is **POS** is generated by

```
(LIST '(TPOP{I}_<ZN>_Y)                                    ;1
      '(MOVE_X_*)                                          ;2
      (TAG 'PC (CONS (NAME PROGRAM) (ADD1 PCN)))           ;3
      '(JUMP-N_X)                                          ;4
      '(JUMP-Z_X)                                          ;5
      '(JUMP_*)                                            ;6
      (PC (CADDR INS) PROGRAM)).                           ;7
```

This code may be read as follows: (1) Pop the stack into **y** and set both the zero and negative flags. (2-3) Move into **x** the address of the beginning of the next Piton instruction. (4) Jump to the address in **x** if the negative flag is on. (5) Jump to the address in **x** if the zero flag is on. (6-7) Jump unconditionally to the label in the **TEST-INT-AND-JUMP** instruction. The claim is that this code jumps to the indicated label iff the top of the stack is positive. The proof of correctness requires the (trivial) fact that an integer is positive iff it is neither zero nor negative.

More interesting is the i-code for the **LT-INT** instruction. It is supposed to determine if the top two elements of the stack are in the ‘‘less than’’ relation (where the deeper element is the lesser). This instruction is supposed to work correctly for any two representable integers. The i-code is

```
'((TPOP_X)                                                     ; 1
  (SUB_<NV>_<TSP>{I}_X{I})                                     ; 2
  (MOVE_<TSP>_*)                                               ; 3
  (BOOL F)                                                     ; 4
  (MOVE-V_<TSP>_*)                                             ; 5
  (BOOL T)                                                     ; 6
  (MOVE_X_*)                                                   ; 7
  (BOOL F)                                                     ; 8
  (MOVE-N_X_*)                                                 ; 9
  (BOOL T)                                                     ;10
  (XOR_<TSP>{B}_X{B})).                                        ;11
```

This code (1) pops the stack into **X** and then (2) subtracts **X** from the new top of the stack, storing the result on top of the stack and setting both the negative and the overflow flags. Instruction (3-4) writes **F** to the top of the stack and instruction (5-6) overwrites it with **T** if if the overflow flag is on. Instruction (7-8) puts an **F** in **X**, which is overwritten with **T** by instruction (9-10) if the negative flag is set. Finally, at (11) the code computes the exclusive-or of the top of the stack and **X**, leaving the result on the top of the stack. The claim is: the stack has been popped twice and a **T** or an **F** has been pushed according to whether the two popped elements were in the required less than relation. This is a non-trivial claim and depends upon the fact that after computing i-j in the twos-complement representation, the exclusive-or the negative and overflow flags is equivalent to i<j.

The hardest instruction to handle was, of course, **CALL**, where an arbitrary number of i-code instructions must be executed. The proof of the one-way correspondence lemma for **CALL** required two inductively proved lemmas, one to show that the execution of the code generated by **GENERATE-PRELUDE1** correctly pushes the initial values of the temporaries and the other to show that the execution of the code generated by **GENERATE-PRELUDE2** correctly pushes the actual values of the formals and removes them from the temporary stack.

It was the need to cope with the hidden resources that caused us to introduce the **R**-level machine. At first sight the **R**-level machine is present to let us separate the issue of stack representation from the issue of instruction set. This is certainly a useful service performed. Had we tried to do the proof of the correctness of the compiler directly from the **P**-level down to the **I**-level, it would have been complicated by the need both to grapple with the representation of stacks as arrays and registers while simultaneously grappling with the change of instruction set. But that simplification was not the driving force behind the introduction of **R**. The main contribution of the **R**-level machine is that it separates the hidden resources from the visible ones and it establishes that the hidden ones can be used arbitrarily within the ‘‘basic block’’ of a Piton instruction. We elaborate this point below.

Consider the attempt to go directly from the **P** level to the **I** level via what we shall call **P->I**.

```
                 n P steps

      P -------------------> *
      |                      |
      |                      |
 P->I |                      | P->I
      |                      |
      |                      * i1
      *--------------------> * i2

                 k I steps
```

It is not the case that i_1 is the same as i_2, because of hidden resources. Tackling the problem as before, we introduce the notion of ‘‘i-equivalence,’’ which checks that the visible part of two i-states are equal. The one-way correspondence formula relating **P** to **I** states the i-equivalence of i_1 and i_2. The proof requires induction on the number, n, of Piton instructions executed. The induction hypothesis establishes that the i-states are i-equivalent after n-1 Piton steps. We need to prove the i-equivalence for n Piton steps. The key is what might be called the ‘‘i-equivalence congruence’’ lemma, which states that i-equivalent states are produced by running the **I** machine on i-equivalent states. But this relationship is not valid!

A computation on the **I** machine can distinguish two i-equivalent states. More precisely, it is possible to step from two i-equivalent states to two states that are not i-equivalent. For example, suppose the instruction to be executed is **TPUSH_X**, which pushes onto the temporary stack the contents of the **X** register. Note that this instruction moves the contents of an invisible resource into a visible one. We call this *exposing* the invisible resource. The **TPUSH_X** instruction exposes **X** and if two i-equivalent states differ on **X**, then the execution of **TPUSH_X** produces non-i-equivalent states.

However, consider a block of i-code in which **X** is initialized from visible resources and then a **TPUSH_X** is done. The execution of that block of i-code in arbitrary i-equivalent states produces i-equivalent states, even though **TPUSH_X** exposes invisible resources. The i-code for each Piton instruction has the property that the hidden resources are all set before they are referenced—i.e., the hidden resources are all treated as temporaries within each block of i-code. The ‘‘i-equivalence congruence’’ lemma could be restated correctly by restricting our attention to i-code sequences with this property. However, it is difficult (though not impossible) to characterize such sequences.

We leapt over the whole question of such defining such sequences by introducing the **R** machine. In one clock tick the **R** machine carries out one Piton instruction and side-effects all the hidden resources appropriately. Proving the r-equivalence congruence lemma is straightforward because the **R** machine does not reference the hidden resources, it only sets them. When we move down from **R** to **I**—running Piton instructions at the **R**-level and running the generated i-code at the **I**-level—we find the state diagram actually commutes—i.e., we do not need i-equivalence but can use equality—because the **R** machine sets the invisible resources exactly as set by the generated i-code.

11.4.3. I->M

We now move down one step further and relate the **I** machine to the **M** machine. Recall that the **I** machine executes symbolic i-code on tagged data objects. The **M** machine is equivalent to **FM8502**; its memory and registers contain only bit vectors. The relation between **I** and **M** is explained with **I->M**, which is the link-assembler.

Theorem. ONE-WAY-CORRESPONDENCE-I-M
```
(IMPLIES (AND (EQUAL (I-PSW (I I N)) 'RUN)
              (EQUAL (I-WORD-SIZE I) 32))
         (EQUAL (I->M (I I N))
                (M (I->M I) N)))
```

If we simply read **M** as **FM8502**, this theorem says that the FM8502 image of any non-erroneous **I**-level computation from some initial i-state can be alternatively obtained by running FM8502 on the image of the initial state. Note that there is no sense of hidden resources or basic blocks of i-code here. We prove that the image of any i-code program executes correctly on FM8502.

Here, as in the **P->R** proof, we see each machine taking the same number of steps. The proof of this one-way correspondence theorem thus breaks down immediately to the proof of the single step version for each machine, and that in turn breaks down to the case for each i-code instruction.

Below we exhibit the lemma for the i-code instruction **ADD_<TSP>{N}_X{N}**.

Theorem. **ADD_<TSP>{N}_X{N}-ONE-WAY-CORRESPONDENCE-I-M-STEP**

```
(IMPLIES (AND (I-STATE-OKP I)
              (EQUAL (I-PSW I) 'RUN)
              (EQUAL (I-WORD-SIZE I) 32)
              (I-ADD_<TSP>{N}_X{N}-OKP I)
              (EQUAL (I-CURRENT-INSTRUCTION I) '(ADD_<TSP>{N}_X{N})))
         (EQUAL (I->M (I-ADD_<TSP>{N}_X{N}-STEP I))
                (M-STEP (I->M I))))
```

To prove this theorem, consider first the left-hand side of the conclusion. Symbolically expanding **(I-ADD_<TSP>{N}_X{N}-STEP I)** produces an i-state in which the topmost element of the temporary stack has been replaced by the tagged natural number obtained by summing the (untagged) old value on the stack and the (untagged) value of register **x**. Then symbolically evaluating the **I->M** on that i-state produces an FM8502 core image. The key property of this core image is that the bit vector at the address pointed to by register 3 (the **tsp** register) is the binary representation of the natural number sum described above.

Now consider the right-hand side of the conclusion, **(M-STEP (I->M I))**. The first issue that must be faced in this proof is to determine what is the current instruction in the m-state **(I->M I)**, given that the current instruction in the i-state **I** is **(ADD_<TSP>{N}_X{N})**. The answer is that it is the bit vector obtained by calling **LINK-INSTR-WORD** on **(ADD_<TSP>{N}_X{N})**, i.e., the bit vector corresponding to the assembly instruction **(ADD NIL (TSP) X)**, which is **B00000000000000110000000101100100**. Then one can symbolically execute **M-STEP**, the single stepper for the **M** machine (FM8502). The result is a new m-state in which the bit vector at the address in register 3 (**B011**, the **tsp** register) is now the ‘‘binary-sum’’ of the old bit vector in that location and the contents of register 4 (**B100**, the **x** register). By ‘‘binary-sum’’ we mean the bit vector produced by the arithmetic-logical unit when the opcode is **B00110**.

We see that the left-hand and right-hand m-states are equivalent if we simply know that the binary representation of the sum of two naturals is the binary-sum of the binary representations of the two naturals. This is true if the sum is representable, which is assured by the **-OKP** predicate above.

In general the key lemmas that had to be proved to construct the **I->M** level proofs had to do with

- the correspondence between the current instruction at the **I**-level and the current instruction at the **M**-level, as illustrated above;
- the commutativity of the fundamental data type operators and the bit-vector representations, as illustrated above; and
- the correspondence between name-offset address pairs into a structured memory segment at the **I**-level and the bit-vector addresses into a linear memory at the **M**-level.

The latter issue was actually involved in the illustration above. For example, when we fetched the current instruction of the m-state, we first inspected the program counter (register 0) and found there a bit-vector. That bit-vector, produced by **I->M**, was constructed by linking the data word which was the program counter at the **I**-level. That data word was an address pair that named a program and an offset. It was linked by computing a natural number indicating the analogous location in the linked memory and then converted to a bit-vector. When the **M** machine fetches from that bit-vector address it gets the bit-vector produced by linking the instruction word found in the i-state at the address indicated by the original address pair.

Perhaps the most interesting instruction to prove correct at this level was the trivial i-code instruction **(INT-TO-NAT)**, which is the only instruction used in the compiled version of the Piton instruction **(INT-TO-NAT)**. At the i-code (and Piton) level, **(INT-TO-NAT)** pops an integer off the stack and pushes the corresponding natural, provided the integer was non-negative. At the machine code level, **(INT-TO-NAT)** is mapped into a no-op.

The one-way correspondence lemma for that instruction is proved by the following steps. On the left-hand side, the **I**-level step retags the top of the stack from **INT** to **NAT** and then the **I->M** maps it down, mapping the top of the stack as though it were an integer (since it is so tagged). On the the right-hand side, **I->M** maps down the original top of the stack as though it were a natural (since it is so tagged) and then the **M**-level step does nothing. The two m-states are equal because the bit-vector representing a small non-negative integer is the same as that representing the same natural.

11.5. The Correctness Proof

In this section we prove the correctness of the FM8502 implementation of Piton. We have already displayed four of the key lemmas, the one-way correspondence theorems relating the adjacent pairs of machines. We display them again below and give them each a number for future reference. During the proof of the correctness result we will mention several other theorems but we will not discuss their proofs.

Theorem 1. ONE-WAY-CORRESPONDENCE-P-R
```
(IMPLIES (AND (PROPER-P-STATEP P)
              (P-LOADABLEP P)
              (NOT (ERRORP (P-PSW (P P N)))))
         (R-EQUAL (P->R (P P N))
                  (R (P->R P) N)))
```

Theorem 2. ONE-WAY-CORRESPONDENCE-R-I
```
(IMPLIES (AND (PROPER-R-STATEP R)
              (NOT (ERRORP (R-PSW (R R N)))))
         (EQUAL (R->I (R R N))
                (I (R->I R)
                   (CLOCK R N))))
```

Theorem 3. ONE-WAY-CORRESPONDENCE-I-M
```
(IMPLIES (AND (EQUAL (I-PSW (I I N)) 'RUN)
              (EQUAL (I-WORD-SIZE I) 32))
         (EQUAL (I->M (I I N))
                (M (I->M I) N)))
```

Theorem 4. FM8502-EQUAL-M
```
(IMPLIES (M-STATEP M)
         (EQUAL (FM8502 M N)
                (M M N)))
```

The correctness theorem is

Theorem. FM8502 Piton is Correct

```
(IMPLIES (AND (PROPER-P-STATEP P0)
              (P-LOADABLEP P0)
              (EQUAL (P-WORD-SIZE P0) 32)
              (EQUAL PN (P P0 N))
              (NOT (ERRORP (P-PSW PN)))
              (EQUAL TS (TYPE-SPECIFICATION
                            (P-DATA-SEGMENT PN))))
         (EQUAL (P-DATA-SEGMENT PN)
                (DISPLAY-M-DATA-SEGMENT
                         (FM8502 (LOAD P0)
                                 (FM8502-CLOCK P0 N))
                         TS
                         (LINK-TABLES P0))))).
```

By expanding the occurrences of **LOAD**, **FM8502-CLOCK**, and **LINK-TABLES** according to their definitions, and replacing **TS** and **PN** by the terms to which they are equated in the hypothesis this theorem reduces to

Theorem. FM8502 Piton is Correct

```
(IMPLIES (AND (PROPER-P-STATEP P0)
              (P-LOADABLEP P0)
              (EQUAL (P-WORD-SIZE P0) 32)
              (EQUAL PN (P P0 N))
              (NOT (ERRORP (P-PSW (P P0 N))))
              (EQUAL TS (TYPE-SPECIFICATION
                            (P-DATA-SEGMENT (P P0 N)))))
         (EQUAL (P-DATA-SEGMENT (P P0 N))
                (DISPLAY-M-DATA-SEGMENT
                         (FM8502 (I->M (R->I (P->R P0)))
                                 (CLOCK (P->R P0) N))
                         (TYPE-SPECIFICATION
                          (P-DATA-SEGMENT (P P0 N)))
                         (I-LINK-TABLES (R->I (P->R P0))))))).
```

Therefore, assume **P0** is a loadable proper p-state with word size 32 that when run forward **N** steps at the **P** level produces a non-erroneous state **PN**.

Using these assumptions and the four correspondence theorems above we draw the following diagram to name a collection of states. For example, the diagram establishes the convention that **PN** is **(P P0 N)**, **R0** is **(P->R P0)**, **RN** is **(R R0 N)**, **RN'** is **(P->R PN)** and that **RN** and **RN'** are **R-EQUAL**.

```
                     P^N
       P0 -------------------------> PN
       |                              \
       |                               \
  P->R |                                \  P->R
       |                                 \
       |                                 |
       |             R^N                 |
       R0 -------------------------> RN  RN'
       |                              \        R-EQUAL
       |                               \
  R->I |                                \     R->I
       |                                 \
       |             I^C                  |
       I0 ------------------------------> Ic
       |                                   |
  I->M |                                   |  I->M
       |                                   |
       |                                   |
       |             M^C                   |
       M0 ------------------------------> Mc
        \                                 /
         \        FM8502^C               /
          ------------------------------
```

To interpret the diagram, **c** should be read as **(CLOCK (P->R P0) N)**.

Observe that the diagram suggests two equivalent definitions of **Ic**: it is the result of running **R0** forward **N** steps and mapping down with **R->I**, and it is the result of mapping **R0** down with **R->I** and then running that state **c** steps forward at the **I** level.

To derive this diagram from the above theorems one must know certain other lemmas. In some sense the topmost box above captures Theorem 1 (**ONE-WAY-CORRESPONDENCE-P-R**, page 219) and the box below that captures Theorem 2 (**ONE-WAY-CORRESPONDENCE-R-I**, page 221). But nothing we have mentioned so far allows us to draw the boxes so that they share the r-states **R0** and **RN**. That is, the suggestion that **Ic** may be derived by either of the two routes shown above is valid only if we know that the hypotheses of Theorem 2 are true when we instantiate its high-level states with the lower level states of Theorem 1.

More precisely, we can usefully compose Theorems 1 and 2 only if we know that the image of **P0** under **P->R**, i.e., **R0**, is a proper r-state and that **R0** can be run at the **R**-level non-erroneously. The first we get from

Theorem 5. PROPER-P-STATEP-IMPLIES-PROPER-R-STATEP
```
(IMPLIES (AND (PROPER-P-STATEP P)
              (P-LOADABLEP P))
         (PROPER-R-STATEP (P->R P))).
```

The second we get from a corollary of Theorem 1 (**ONE-WAY-CORRESPONDENCE-P-R**, page 219), namely that the **R-PSW** of **(R (P->R P) N)** is the same as the **P-PSW** of **(P P N)**. This follows from the definitions of **R-EQUAL** and **P->R**.

The third box above captures Theorem 3 (**ONE-WAY-CORRESPONDENCE-I-M**, page 226). That theorem has two hypotheses also. The first requires that the initial i-state, **I0** in the diagram, have word size 32, which follows from the assumption that **P0** has word size 32 and thus so does **R0** and **I0** (given the definitions of our mapping functions). The second requires that the psw of the final i-state, **Ic** in the diagram, is **'RUN**, which follows from the definition of **R->I** and the previous observation that **RN** is non-erroneous.

Finally, the fourth box above captures Theorem 4 (**FM8502-EQUAL-M**, page 218), which requires only that the initial state, **M0** in the diagram, be an m-state. This follows from the definition of **I->M**.

Thus, we claim that we are justified in using the diagram above to establish naming conventions for the various states.

We are interested in **(P-DATA-SEGMENT PN)**. But this is equal to **(R-USR-DATA-SEGMENT RN')** (by the definition of **P->R**), which in turn is equal to **(R-USR-DATA-SEGMENT RN)** (by the **R-EQUAL**ity of **RN** and **RN'** and the definition of **R-EQUAL**), which is equal to **(I-USR-DATA-SEGMENT Ic)** (by the definition of **R->I**).

Using these observations and the names established by the diagram, the conclusion of the main theorem becomes

```
(EQUAL (I-USR-DATA-SEGMENT Ic)
       (DISPLAY-M-DATA-SEGMENT
                        (I->M Ic)
                        (TYPE-SPECIFICATION (I-USR-DATA-SEGMENT Ic))
                        (I-LINK-TABLES I0))).
```

Running the **I** machine does not affect the link tables computed for an i-state, i.e.,

Theorem 6. I-LINK-TABLES-I

```
(IMPLIES (AND (EQUAL (I-PSW (I I0 C)) 'RUN)
              (EQUAL (I-WORD-SIZE I0) 32))
         (EQUAL (I-LINK-TABLES (I I0 C))
                (I-LINK-TABLES I0))).
```

Thus, **(I-LINK-TABLES I0)** is the same as **(I-LINK-TABLES Ic)**. The hypotheses of Theorem 6, above, have already been shown to hold of our particular **I0** and **C**.

Thus, the conclusion of the main result is

```
(EQUAL (I-USR-DATA-SEGMENT Ic)
       (DISPLAY-M-DATA-SEGMENT
                        (I->M Ic)
                        (TYPE-SPECIFICATION (I-USR-DATA-SEGMENT Ic))
                        (I-LINK-TABLES Ic))).
```

The proof is then completed by appealing to the following theorem, which essentially says that **DISPLAY-M-DATA-SEGMENT** inverts **I->M** on the data segment:

Theorem 7. **DISPLAY-M-DATA-SEGMENT-INVERTS-I->M**

```
(IMPLIES (AND (PROPER-I-USR-DATA-SEGMENTP (I-USR-DATA-SEGMENT I) I)
              (I-STATE-OKP I))
         (EQUAL
          (DISPLAY-M-DATA-SEGMENT
           (I->M I)
           (TYPE-SPECIFICATION (I-USR-DATA-SEGMENT I))
           (I-LINK-TABLES I))
          (I-USR-DATA-SEGMENT I))).
```

It remains only to establish the two hypotheses of Theorem 7. The first is that the user data segment of **Ic** is proper. But **Ic** is the same state as **(R->I RN)** and so its user data segment is proper, by

Theorem 8. **PROPER-R-STATEP-IMPLIES-PROPER-I-USER-DATA-SEGMENTP**

```
(IMPLIES (PROPER-R-STATEP R)
         (PROPER-I-USR-DATA-SEGMENTP (R-USR-DATA-SEGMENT R)
                                     (R->I R))),
```

provided we could show that **RN** is a proper r-state. We have already argued that **R0** is proper and it turns out therefore that **RN** is proper because proper r-states are preserved by **R**,

Theorem 9. **PROPER-R-STATEP-R**

```
(IMPLIES (PROPER-R-STATEP R)
         (PROPER-R-STATEP (R R N))).
```

The second hypothesis of Theorem 7 is **(I-STATE-OKP Ic)**. (**I-STATE-OKP** is the **I**-level analogue of the proper state invariant.) The following theorem establishes that proper r-states map down to proper i-states:

Theorem 10. **PROPER-R-STATEP-IMPLIES-I-STATE-OKP-R->I**

```
(IMPLIES (AND (PROPER-R-STATEP R)
              (NOT (ERRORP (R-PSW R))))
         (I-STATE-OKP (R->I R))).
```

We do not discuss the proofs of Theorems 5-10. All of the theorems cited have been proved mechanically.

Appendix I. Primitive Functions

The following primitive function symbols are used in this document: **ADD1**, **AND**, **APPEND**, **ASSOC**, **CAR**, **CDR**, **CONS**, **DIFFERENCE**, **EQUAL**, **IF**, **IMPLIES**, **LESSP**, **LISTP**, **LITATOM**, **MEMBER**, **MINUS**, **NEGATIVE-GUTS**, **NEGATIVEP**, **NLISTP**, **NOT**, **NUMBERP**, **OR**, **PACK**, **PAIRLIST**, **PLUS**, **QUOTIENT**, **REMAINDER**, **STRIP-CARS**, **SUB1**, **TIMES**, **TRUEP**, **UNPACK**, **ZERO** and **ZEROP**.

See [4] for the axioms defining these functions.

Appendix II. Statistics

II.1. History of the Project

The project originally started in September, 1986. At that time, Warren Hunt, of Computational Logic, Inc., and the author sketched out a stack based assembly language for FM8501 and implemented an assembler and linker for a small subset of it containing about 10 instructions. This was done without defining the formal semantics of the language; such an assembler would be merely a convenient way to produce machine code. Properties of the machine code programs would be proved directly from the FM8502 definition. This view of an assembler level language is exactly that taken by Bevier in [1]. If the machine code programs thus produced can overwrite themselves, as is possible with all conventional assemblers, this is the only view possible. The instructions of the program *must* be bit vectors rather than symbolic expressions since data can be treated as instructions.

The desire to prove theorems about our programs at a higher level than the FM8501 definition forced us to define the semantics formally and lift the language somewhat. Hunt and the author then designed a ‘‘toy language’’ that contained only four instructions: a simplified **CALL** (with no provision for formals), **RET**, a variable-to-variable **MOVE**, and an increment-by-2 instruction, **ADD2**. This language was called **H** (for **H**igh level). We defined a 10 instruction low-level machine, called **L** which was a simplified FM8501, on which it was just possible to implement **H**. We implemented **H** via a compiler and link-assembler and formulated what was meant by the correctness of the implementation. During this period we came up with the idea of the psw and erroneous states.

We then began the task of trying to prove the correctness of the implementation. Around this time, Hunt began working on other hardware verification efforts and the project became the sole concern of the author.

In the first attempt to prove the correctness of the **L** implementation of **H**, a single intermediate machine was introduced, comparable to our **I**. The proof from **I** down to **L** was completed and the proof from **H** to **I** was begun. Soon afterwards we recognized the ‘‘hidden resource’’ problem and introduced the **R** machine, **R-EQUAL** and the congruence properties.

The entire proof of the correctness of the **L** implementation of **H** was completed by September, 1987, a year after we began work on the assembly language design. During the first seven months of that year, the project was staffed by 2 men working roughly 8 hours per week. During the last 5 months, the project was staffed by 1 man working roughly 8 hours a day, less about one month of time off. Thus, 7 man-months were devoted to what might be called the ‘‘Piton feasibility study.’’ The importance of this early phase of the project cannot be overemphasized. The identification of the hidden resource problem and the formal methods for coping with it (the **R** machine and **R-EQUAL**) would have been enormously more costly had it occurred in the middle of the Piton project.

Work on the full-blown language, implementation and proof began in September, 1987. This work was done by the author alone.

The style of the Piton language definition was heavily influenced by the experience with **H**. This is not to say that the definition was in the style of **H**! Rather, in the definition of Piton we tried to avoid the mistakes we had made in the definition of **H**. One of the main improvements was the isolation of each instruction's **-OKP** and **-STEP** function so that the language could grow very large without complicating the proof. Another was the uniform handling of the psw and errors. Many new issues had to be addressed, including the handling of multiple data types, type checking, and the role of the **TYPE-SPECIFICATION** in the correctness result.

It was also decided to define the **R** and **I** machines as part of the initial specification process. While these intermediate machines were not technically necessary for the implementation we knew they were needed for the proof and (accurately) felt that their definitions would serve as valuable specifications for the internal forms to be produced by the implementation.

Bill Young of Computational Logic, Inc., and a graduate student in the Computer Sciences Department of the University of Texas at Austin, began using a simplified version of the evolving Piton as the target language for a MicroGypsy compiler. Young's requirements caused several changes in Piton during the course of its proof, necessitating the modification of previously completed modules. In December Matt Kaufmann, of Computational Logic, Inc., volunteered do some of the Piton proofs with his interactive enhancement to the Boyer-Moore theorem prover.

II.2. Manpower Requirements

The time-line below shows the progress on the full-blown Piton project. The numbers **1**-**8** below indicate specific subtasks enumerated below.

```
                         1987 1988
           Sep  Oct  Nov  Dec  Jan  Feb  Mar  Apr  May
          |    |    |    |    |    |    |    |    |    |

Moore      1111 1111 1122 2222                4445 6678
Kaufmann                  3333 3333           4455
```

Task **1**: Definition of Piton, FM8502, the implementation, the concepts used in the correctness theorem, and all of the intermediate machines.

Task **2**: Proof of the one-way correspondence theorem for the **I->M** step and the equivalence of **FM8502** and **M**.

Task **3**: Proof of the one-way correspondence theorem for the **P->R** step.

Task **4**: Proof of the one-way correspondence theorem for the **R->I** step.

Task **5**: Proof of the other supporting lemmas for the top-level theorem and the proof of the top-level theorem itself.

Task **6**: A new proof of the **P->R** step, after revising the the definition of Piton to distinguish the static preconditions from the dynamic ones. See below.

Task **7**: A consolidation and cleaning up of all of the proofs.

Task **8**: The addition of seven new Piton instructions, **LT-INT**, **PUSH-TEMP-STK-INDEX**, **FETCH-TEMP-STK**, **DEPOSIT-TEMP-STK**, **POP***, **POPN**, and **INT-TO-NAT** and the replaying of all the proofs to accommodate changes in every machine.

During the course of the proof many ‘‘bugs’’ were discovered in the definitions of the various machines and invariants. These bugs sometimes rippled out to other machines and invariants. For example, an omitted precondition in **P** might also be omitted from **R** without jeopardizing the **P->R** step; the **R->I** step might fail because of the omitted check. Such a bug would be found in the **R->I** proof and then require changes to both **R** and **P** and reconsideration of the **P->R** step.

The time line above is misleading because it does not accurately reflect the effort expended in modifying and reconsidering previously ‘‘completed’’ proofs. The fact that we had two proof efforts running in parallel mitigated this problem somewhat. When one of us had to change any definition, the change was communicated to the other and one of us would informally investigate whether that change needed to be propagated to other levels. If the change impacted the other proof effort, the person managing that effort would wait until he got to a nice stopping place, would make the change, reconstruct his proof to that point, and then continue. This was much less expensive (in terms of user ‘‘context switching’’) than it would have been had the proofs been constructed sequentially.

By May the entire proof had been mechanically checked, but some proofs had been done with the Boyer-Moore theorem prover and others had been done by the Kaufmann enhancement (according to whether Moore or Kaufmann had managed the task).

Task **6**, the revision of Piton and the reconstruction of the **P->R** proof, was necessitated by the kind of sloppy iteration described above. By the time the three one-way correspondence theorems had been proved a lot of incremental modifications had been made. These had rippled up to the **P** level and showed up as *ad hoc* restrictions in the dynamic preconditions. Many of these restrictions could have been moved into the static checks enforced by **PROPER-P-STATEP** or were strictly unnecessary because of checks made there. In Task **6** we carefully segregated the static and dynamic checks and pared down the dynamic checks as much as possible. (Since the dynamic checks are part of the Piton interpreter, they complicate proofs *about* Piton programs.)

Because Moore managed Task **6** a side-effect of that task was to carry out the **P->R** step proof with the Boyer-Moore theorem prover. In a narrow sense, all of the lemmas and proof commands developed by Kaufmann in Task **3** were discarded and the new proof was constructed from scratch. But in a much more meaningful sense, the new proof was just a modification of the Kaufmann proof because the earlier proof had debugged the statement of the theorem and gotten correct the hundreds of definitions involved in the **P** and **R** machines.

By the time Task **7** had been completed, the entire proof was constructed by the Boyer-Moore theorem prover.

Task **8** was motivated by accumulated requests from Bill Young for additional language features. We were encouraged by the fact that seven new instructions could be added, necessitating the change of every machine except **FM8502**, and the entire proof reconstructed in less than a week. This was the case primarily because each level of the proof first develops a powerful library of rules for dealing with the concepts at that level and so minor changes to theorems were accommodated automatically. The new instructions were added by choosing a similar old instruction and visiting every occurrence of that old name in the proof event files. For each formula involving the old instruction an analogous formula involving the new instruction was inserted. With minor exceptions the resulting transcripts were automatically processed. When the automatic processing failed it was because of some relatively deep problem specific to the new instruction (e.g., that integer less than can be computed by taking the exclusive-or of the negative and overflow flags after a subtraction).

The experience of adding seven new instructions in less than a week has made us optimistic that Piton can evolve to suit the needs of its users. We cannot yet answer the commonly asked questions ‘‘how hard would it be to implement Piton on a different hardware base and prove that correct?’’ and ‘‘how hard would it be to implement and prove the correctness of a different language?’’ Whatever the answers, we know those problems are made significantly simpler because of our Piton experience.

For what it is worth, a total of 9 man-months was spent on the full-blown Piton project. Nine months elapsed from start to finish. This of course ignores the 7 man-months devoted during the preceding 12 months to the ‘‘feasibility’’ study. This report has taken two man-months to assemble.

II.3. Sizes of the Formal Systems

The following table shows the number of bytes and the number of lines of user-supplied pretty-printed formal text involved in the definition, implementation and proof of Piton.

	bytes	lines
FM8502	49K	1706
Piton	73K	2825
Implementation	38K	1400
Proof	960K	29859

However, these measures are misleading since long identifiers and short lines tend to inflate the counts. In addition, because of the wonders of GNU Emacs key board macros [18], many of the characters allegedly typed were not actually typed at all but were generated from previous type-in. For readers unfamiliar with text editing we give two examples. Recall that the one-way correspondence theorem at each level is proved by proving a single-step version, which in turn is proved via a single-step lemma for each instruction. In the case of the **`P->R`** proof, there are 65 instructions and hence 65 lemmas. The first one was typed manually. The remaining ones were obtained mechanically by commanding the text editor to successively replace the first opcode by each of the other opcodes. In GNU Emacs this operation can be defined as a single command and used repeatedly thereafter. Another way keystrokes are eliminated is by editing the theorem prover’s output on unsuccessful proof attempts. The typical sequence of actions is: a formula is submitted, the proof fails, the user reads the proof attempt and realizes that the theorem prover needs to know a new lemma, and the lemma is then textually constructed by editing the formula in which it will be used. For example, the user may realize ‘‘This (*click*) **`AND`** this (*click*) **`IMPLY`** that this (*click*) is equal to this (*click*),’’ where each ‘‘*click*’’ signifies a keystroke or two that picks up a term on the screen and deposits it into the emerging lemma. The point is that the size of the formula thus constructed is independent of the number of keystrokes. We therefore prefer to count the number of logical ‘‘events’’ (**`ADD-SHELL`**, **`DEFN`**, and **`PROVE-LEMMA`** commands of the theorem-prover) involved in a system.

Below we enumerate several different subsystems within the entire Piton definition, implementation and proof. For each we show the number of shell definitions, function definitions, and lemmas proved ‘‘for’’ that subsystem.

There are a total of 2,764 names involved in the entire system. Every name has been assigned to exactly one of the subsystems in which it is used—though many names are used in many subsystems.

	Shells	Definitions	Theorems
Statement of the Problem			
FM8502	3	78	
Piton	1	320	
Implementation			
Resource Phase	1	11	
Compiler	1	82	
Link-assembler		39	
Load		1	
Totals (Implementation)	(2)	(133)	
Correctness Theorem	__	24	
Totals	6	555	
Proof of Correctness			
Statement of the Problem	6	555	
R Machine		173	
I Machine		197	
M Machine		12	
Theorem 1 (**P->R**)		27	690
Theorem 2 (**R->I**)		74	245
Theorem 3 (**I->M**)		19	385
Theorem 4 (**FM8502-EQUAL-M**)			4
Theorems 5-10	__	23	364
Totals	6	1080	1688

The proof of Theorem 1 is clearly the most difficult. This is intuitively surprising since Theorem 1 only involves a change of representation. Theorems 2 and 3 are much more interesting since they concern compilation and link-assembling. What makes Theorem 1 difficult is that it is dealing with the hidden resource problem.

Recall our discussion of the proof of Theorem 1. We noted two key lemmas, **ONE-WAY-CORRESPONDENCE-P-R-STEP** (page 219), and **R-EQUAL-CONGRUENCE** (page 220), and informally mentioned six others. The proof of **ONE-WAY-CORRESPONDENCE-P-R-STEP** cost 283 lemmas. This cost alone would make the cost of Theorem 1 comparable to that of Theorem 2 and simpler than Theorem 3. Thus, the difficulty of Theorem 1 would correspond intuitively to the idea that it was just a change of representation.

But while all three Theorems 1-3 involve a single-step one-way-correspondence lemma, Theorems 2 and 3 need no other major lemmas while Theorem 1 does. The second key lemma in the proof of Theorem 1 is **R-EQUAL-CONGRUENCE**, which costs 191 lemmas. Observe that this is an inductively proved fact about the lower level machine; in Theorems 2 and 3 the only facts needed about the lower level machines are how to run them on symbolic data. The six other lemmas involved in Theorem 1 have a combined cost of 216 lemmas and involve such complicated observations as that proper p-states map down to proper r-states and that the **R** machine preserves proper r-states. Unlike the proofs of Theorems 2 and 3, the proof of Theorem 1 intimately involves invariants about the lower level machine, not just invariants about the upper level machine (i.e., that proper states are preserved by the upper level single stepper). These invariants primarily concern resource hiding.

The total time taken to reproduce the proofs on a 12 megabyte Sun 3/60 running the Austin Kyoto Common Lisp implementation of the Boyer-Moore theorem prover [4] is 16 hours.

References

1. W. Bevier. *A Verified Operating System Kernel*. Ph.D. Th., University of Texas at Austin, 1987.

2. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

3. R. S. Boyer and J S. Moore. A Verification Condition Generator for FORTRAN. In *The Correctness Problem in Computer Science*, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

4. R. S. Boyer and J S. Moore. A User's Manual for A Computational Logic. Tech. Rept. Technical Report 18, Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703, 1988.

5. Dan Craigen. *A Description of m-Verdi [Working Draft].* I. P. Sharp Associates, Ltd., 1986.

6. S. L. Gerhart, D. R. Musser, D. H. Thompson, D. A. Baker, R. L. Bates, R. W. Erickson, R. L. London, D. G. Taylor and D. S. Wile. An Overview of AFFIRM: A Specification and Verification System. Information Processing 80, S. H. Lavington (Ed.), October, 1980, pp. 343-348. North Holland Publishing Company.

7. Donald I. Good. Mechanical Proofs about Computer Programs. In C. A. R. Hoare and J. C. Shepherdson, Ed., *Mathematical Logic and Programming Languages*, Prentice-Hall International Series in Computer Science., 1985, pp. 55-75.

8. Donald I. Good, Robert L. Akers, Lawrence M. Smith. *Report on Gypsy 2.05 - January 1986.* Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703., 1986.

9. Michael K. Smith, Donald I. Good, Benedetto L. DiVito. *Using the Gypsy Methodology.* Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703., 1988. Revised January 1988.

10. Mike Gordon. Proving a Computer Correct. Tech. Rept. TR 42, University of Cambridge, Computer Laboratory, 1983.

11. Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor*. Ph.D. Th., University of Texas at Austin, 1985.

12. P. M. Melliar-Smith and R. Schwartz. Hierarchical Specification of the SIFT Fault-Tolerant Flight Control System. Tech. Rept. CSL-123, Computer Science Laboratory, SRI International, Menlo Park, Ca., 1981.

13. David R. Musser and David A. Cyrluk. *AFFIRM-85 Installation Guide and Reference Manual Update.* General Electric Corporate Research and Development, 1985.

14. P. G. Neumann, L. Robinson, K. Levitt, R. Boyer, A. Saxena. A Provably Secure Operating System. Tech. Rept. CSL-116, Computer Science Laboratory, SRI International, 1977.

15. W. Polak. *Compiler Specification and Verification.* Springer-Verlag, Berlin, 1981.

16. L. Robinson and K. Levitt. "Proof Techniques for Hierarchically Structured Programs". *Comm. ACM 20*, 4 (April 1977).

17. Mark Saaltink. The Verdi Logic [Working Draft]. I. P. Sharp Associates, Ltd., 1986.

18. Richard M. Stallman. *GNU Emacs Manual.* Free Software Foundation, 1000 Massachusetts Avenue, Cambridge, MA 02138, 1987.

19. D.F. Stanat, T.A. Thomas, and J.R. Dunham. Proceedings of a Formal Verification/Design Proof Peer Review. Tech. Rept. RTI/2094/13-01F, Research Triangle Institute, P.O. Box 12194, Research Triangle Park, N.C., 27709, 1984.

20. Stanford Verification Group. *Stanford Pascal Verifier User Manual.* Stanford University, 1979.

21. D. Thompson and W. Erikson. AFFIRM Reference Manual. USC Information Sciences Institute, 4676 Admiralty Way, Marina Del Rey, Ca. 90291, 1981.

Index

The numbers associated with each entry of this index are page numbers. Each number is in one of three fonts. Bold face numbers, such as **27** and **135**, indicate the defining occurrence of the symbol or phrase. Numbers in Roman font, such as 27 and 135, indicate significant occurrences of the symbol or phrase in text. Not every occurrence of the symbol or phrase in text is deemed "significant." Numbers in typewriter font, such as `27` and `135`, indicate occurrences of the given symbol in the definitions listed in Chapters 7-10 of this report. Every such occurrence is noted. Such page numbers indicate the beginning of the containing definition, rather than the page on which the occurrence is found.

A

C

F

G

I

J

L

M

N

O

P

S

Table of Contents

List of Figures