The Partial Specification of Microprocessor Instruction Set Architectures

William R. Bevier

Technical Report 51
November 1989

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work was sponsored 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 authors 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.

1. Introduction

The purpose of the work described in this paper is to formally specify an instruction set architecture in a way that avoids over-specification. We wish our specification to permit reasonable implementation alternatives. For instance, our instruction set specification does not require a particular instruction format. Relaxation of requirements extends to machine size. We do not require a particular register file length, for example. The result is a specification method not for a single architecture, but for a class of instruction set architectures.

In this paper we describe our approach to specification, giving examples which are drawn from our formal specification for an instruction set for a RISC architecture. Our specification is written in the Boyer-Moore logic [Boyer & Moore 88]. Appendix A contains a brief summary of the Boyer-Moore logic.

1.1 Overview of the Approach

A machine instruction is modeled as a function **`I`** which takes a machine state to a machine state. If **`P`** is a predicate which recognizes a valid machine state, we expect the following theorem to hold on **`P`** and **`I`**.

```
(IMPLIES (P X) (P (I X)))
```

In our approach to instruction set specification, we do not define **`P`** or **`I`**. We instead constrain them to satisfy certain properties. The instruction set specification consists of a set of constraints: some on the machine state, and others on machine instructions. The specification therefore defines a class of machines. It is possible to prove that a fully specified architecture satisfies the constraints, and is therefore an instance of the class of machines. Theorems proven about a class of machines also apply to any instance it.

This formalization can be done in the Boyer-Moore logic with a new feature called *functional instantiation* [Boyer, Goldschlag, Kaufmann, and Moore 89]. The feature permits two new actions. **`CONSTRAIN`** declares a new function symbol and its arguments, and states properties of the function without defining the function. To prove that introduction of the function symbol does not create an inconsistency there is an obligation to exhibit a witness to the constrained function that satisfies the constraints. The action **`FUNCTIONALLY-INSTANTIATE`** allows one to establish that a given function **`G`** is an instance of the class of functions described by a given functional variable **`H`**.

The Boyer-Moore theorem prover provides mechanical support for these features. For a **`CONSTRAIN`** event, it automatically generates the formulas necessary to prove that the witness function satisfies the constraints, and it attempts the proof of these formulas. The event is successful when all of the formulas are proved. For a **FUNCTIONALLY-INSTANTIATE** event, the prover automatically generates the formulas required to prove that a function is an instance of a functional variable, and attempts their proof. Again, the event is successful when all the formulas are proved. The implementation of functional instantiation guarantees the consistency of the theory being developed.

The remaining sections of this paper are organized as follows. Section 2 outlines some primitive functions necessary for specifying machine states and instructions. Section 3 shows how we specify an instruction set architecture by constraints. Section 4 demonstrates how we formally prove that a particular architecture satisfies the constraints.

2. Formal Preliminaries

In this section we introduce some primitive functions necessary to describe machine resources. Three data types are used: booleans, bit vectors, and lists of bit vectors. They are used, respectively, to represent bits, bit strings and arrays of bit strings (e.g., memories).

2.1 Bits

Individual bits are represented by the boolean data type. **T** represents 1 or ON, **F** represents 0 or OFF.

2.2 Bit Vectors

Strings of bits (e.g., bytes, words) are represented by *bit vectors*. A bit vector can be though of as a list (of arbitrary length) whose elements are restricted to the boolean values **T** and **F**. A bit vector is defined in the Boyer-Moore logic by the shell **BV**. This axiomatization follows Hunt's [Hunt 87].

```
(ADD-SHELL BV BV-NIL BVP
           ((BV-BIT (ONE-OF FALSEP TRUEP) FALSE)
            (BV-VEC (ONE-OF BVP) BV-NIL)))
```

Addition of the shell **BV** introduces five new functions: **BV**, **BVP**, **BV-NIL**, **BV-BIT** and **BV-VEC**. The following axioms are asserted about these function symbols. (Others not mentioned are asserted as well.)

```
(BVP (BV-NIL))
(BVP (BV A B))
(EQUAL (BV-BIT (BV A B)) A)
(EQUAL (BV-VEC (BV A B)) B)
```

(BV-NIL) represents the empty bit vector. **BV** contructs a bit vector from a bit and another bit vector. For example, the expression **(BV F (BV-NIL))** represents the bit string "0", and

(BV F (BV T (BV F (BV T (BV-NIL)))))

represents the bit string "0101".

The function **BVP** recognizes a bit vector. The formula **(BVP (BV-NIL))** is true. The function **BV** applied to any arguments satisfies **BVP**. That is, **(BVP (BV A B))** is true for any **A** and **B**. **BVP** applied to a value not constructed from an application of either **BV-NIL** or **BV** is false. For example, **(BVP 1)** is false.

BV-BIT accesses the first bit of a bit vector. For example,

(BV-BIT (BV F (BV T (BV-NIL))))

equals **F**. **BV-VEC** accesses the remainder of a bit vector following the first bit. The expression

(BV-VEC (BV F (BV T (BV-NIL))))

equals **(BV T (BV-NIL))**.

The following defined functions apply to bit vectors. **BV-NILP** recognizes a value which is either the constant **(BV-NIL)** or is not a bit vector. The function **BV-LENGTH** returns the length of a bit vector.

```
(DEFN BV-NILP
      (A)
      (IF (BVP A) (EQUAL A (BV-NIL)) T))

(DEFN BV-LENGTH
      (BV)
      (IF (BV-NILP BV)
          0
          (ADD1 (BV-LENGTH (BV-VEC BV)))))
```

The function **BV-WORDP** recognizes a bit vector of a given length. For example, If **X** is an object for which **(BV-WORDP X 8)** holds, then **X** represents a byte. (The function **FIX** coerces a non-number to **0**.)

```
(DEFN BV-WORDP
      (X BVLENGTH)
      (AND (BVP X)
           (EQUAL (BV-LENGTH X) (FIX BVLENGTH))))
```

The functions **BV-TO-NAT** and **BV-TO-INT** give the unsigned and 2s-complement interpretation, respectively, of a bit-vector. These functions treat the first argument of a bit vector as the least significant bit. For instance, the expression

(BV-TO-NAT (BV T (BV T (BV F (BV T (BV-NIL))))))

equals **11** (in decimal). For the 2s-complement interpretation of a bit vector, the high order bit (innermost in the **BV** representation) is the sign bit. The expression

```
(BV-TO-INT (BV T (BV T (BV F (BV T (BV-NIL))))))
```

equals **-5**. The functions **NAT-TO-BV** and **INT-TO-BV** are the inverses, respectively, of **BV-TO-NAT** and **BV-TO-INT**.

2.3 Lists of Bit Vectors

Arrays of bit strings (e.g., memories, register files) are represented by lists of bit vectors. Lists are a primitive data type in the Boyer-Moore logic, and we do not discuss them at length here. Appendix A introduces lists (i.e., ordered pairs) briefly.

Here are a few essential functions on lists. The symbol **NIL** represents an empty list. **APPEND** appends two lists. **(APPEND ’(A B C) ’(X Y))** equals **’(A B C X Y)**. **LENGTH** returns the length of a list.

The functions **GET** and **PUT** provide direct access to elements of lists. All access uses zero-based indexing. We do not present the definitions of these functions, but instead describe their properties. The form **(GET N L)** returns the **N**th element of a list **L**. **(PUT N V L)** takes as arguments an index **N**, a value **V** and a list **L**. It returns a list identical to **L** everywhere except location **N**, and the value **V** is guaranteed to occur at location **N** if **N** is a valid index (i.e., less than the length of the list).

The lemmas **GET-PUT-COINCIDENCE** and **GET-PUT-NON-INTERFERENCE** state the crucial relationship between **GET** and **PUT**. First, **GET** retrieves a value placed at a given index by **PUT**, where the index is required to be less than the length of the list. Second, **PUT** alters no element of a list other than the one indexed.

```
(PROVE-LEMMA GET-PUT-COINCIDENCE
             NIL
             (IMPLIES (LESSP N (LENGTH L))
                      (EQUAL (GET N (PUT N V L)) V)))

(PROVE-LEMMA GET-PUT-NON-INTERFERENCE
             NIL
             (IMPLIES (NOT (EQUAL (FIX I) (FIX J)))
                      (EQUAL (GET I (PUT J V L))
                             (GET I L))))
```

Lists of bit vectors are used to represent memories. The function **BV-WORDLISTP** recognizes a list of words. Each element of a word list is a bit vector of a given length. The lemma **BV-WORDP-GET** states the fact that getting an element of a word list returns a word.

```
(DEFN BV-WORDLISTP
      (L BVLENGTH)
      (IF (LISTP L)
          (AND (BV-WORDP (CAR L) BVLENGTH)
               (BV-WORDLISTP (CDR L) BVLENGTH))
          T))

(PROVE-LEMMA BV-WORDP-GET NIL
             (IMPLIES (AND (BV-WORDLISTP L BVLENGTH)
                           (LESSP N (LENGTH L)))
                      (BV-WORDP (GET N L) BVLENGTH)))
```

3. Specifying a Class of Machines

In this section we explain how we formally constrain a specification for an instruction set architecture. As an example, we have chosen a simple RISC machine architecture to demonstrate the technique. The architecture is based on the description of a MIPS instruction set given by Firth [Firth 87]. The MIPS architecture was originally designed at Stanford University [Hennessey 82] and a successor is now available commercially. We make no claims that our specification is faithful to any existing MIPS machine.

3.1 Constraining the Machine State

We have the following information about the user-visible state of a MIPS machine from [Firth 87].

- The processor is a 32-bit machine. Words are 32 bits wide, each consisting of four 8-bit bytes.
- Processor resources include at least sixteen 32-bit general purpose registers and a number of special purpose registers. Among the special purpose registers are a program counter and status register, which are assumed to be disjoint from the general purpose registers.
- Fixed point numbers are represented in the 2s-complement notation.
- Memory is byte-addressable, and can be accessed in units of bytes, halfwords and words. There are address alignment requirements for memory access.

We specify the instruction set in terms of its effect on four resources: a register file, a program counter, a status register and a memory. The widths of all words is 32 bits. We do not specify a register file length. Memory is assumed to be a single segment of 32-bit words. The length of memory is not specified. We require that all of these resources are disjoint. The shell **STATE** defines a 4-tuple consisting of these resources. Axioms introduced by the Boyer-Moore shell mechanism guarantee that the resources are disjoint.

```
(ADD-SHELL STATE
           NIL STATEP
           ((REGFILE (NONE-OF) ZERO)
            (PC (NONE-OF) ZERO)
            (SR (NONE-OF) ZERO)
            (MEMORY (NONE-OF) ZERO)))
```

The shell declaration places no type restrictions on the components of a machine state. We introduce a function symbol, **GOOD-STATE**, to constrain the type of each component. The constraints on **GOOD-STATE** are given by the **CONSTRAIN** event **GOOD-STATE-CONSTRAINTS** below. A constraint is a list of the form **(CONSTRAIN <name> <type> <form> (...(<new$_i$> <old$_i$>)...))**, where **<name>** is a symbol which is the name of the event, **<type>** is a directive to the theorem prover which we can ignore, each **<new$_i$>** is a new function symbol, each **<old$_i$>** (the witness function for **<new$_i$>**) is a previously introduced function symbol or a lambda-expression consisting of previously introduced function symbols, and **<form>** (the constraint) is a formula presumably with occurences of the **<new$_i$>**. The **CONSTRAIN** event is successful if **<form>**, with **<old$_i$>** substituted for **<new$_i$>**, is a theorem. See [Boyer, Goldschlag, Kaufmann, and Moore 89] for details.

The constraints on **GOOD-STATE** say three things.

- The type of **GOOD-STATE**: boolean.
- What can be inferred from **GOOD-STATE**: The register file is a list of bit vectors, at least 16 long, each element of which is 32 bits wide. The program counter is 32 bits wide. The status register is 32 bits wide. The memory is a list of bytes, whose length is divisible by 4.
- Necessary conditions to infer **GOOD-STATE**. That is, if we know the above facts about the register file, program counter, status register and memory, then **GOOD-STATE** can be inferred.

As a witness to the constraint, we supply a function which recognizes a state in which the size of the register file is exactly **16** and in which the length of memory is **0**. In response to the constrain event, the Boyer-Moore theorem prover automatically generates the formulas necessary to establish that the witness function satisfies the constraints. The constraint is successfully processed when the formulas are proved, assuring the consistency of the axioms introduced for the new function symbol **GOOD-STATE**.

```
(CONSTRAIN
 GOOD-STATE-CONSTRAINTS NIL
 (AND (OR (TRUEP (GOOD-STATE S))
          (FALSEP (GOOD-STATE S)))
      (IMPLIES (GOOD-STATE S)
               (AND (STATEP S)
                    (BV-WORDLISTP (REGFILE S)
                                  (WORD-LENGTH))
                    (EQUAL (LESSP (LENGTH (REGFILE S)) 16)
                           F)
                    (BV-WORDP (PC S) (WORD-LENGTH))
                    (BV-WORDP (SR S) (WORD-LENGTH))
                    (BV-WORDLISTP (MEMORY S)
                                  (BYTE-LENGTH))
                    (EQUAL (REMAINDER (LENGTH (MEMORY S)) 4)
                           0)))
      (IMPLIES (AND (GOOD-STATE STATE)
                    (STATEP S)
                    (BV-WORDLISTP (REGFILE S)
                                  (WORD-LENGTH))
                    (EQUAL (LENGTH (REGFILE S))
                           (LENGTH (REGFILE STATE)))
                    (BV-WORDP (PC S) (WORD-LENGTH))
                    (BV-WORDP (SR S) (WORD-LENGTH))
                    (BV-WORDLISTP (MEMORY S)
                                  (BYTE-LENGTH))
                    (EQUAL (LENGTH (MEMORY S))
                           (LENGTH (MEMORY STATE))))
               (GOOD-STATE S)))
 ((GOOD-STATE
            (LAMBDA (S)
                    (AND (STATEP S)
                         (BV-WORDLISTP (REGFILE S)
                                       (WORD-LENGTH))
                         (EQUAL (LENGTH (REGFILE S)) 16)
                         (BV-WORDP (PC S) (WORD-LENGTH))
                         (BV-WORDP (SR S) (WORD-LENGTH))
                         (EQUAL (LENGTH (MEMORY S)) 0))))))

(DEFN BYTE-LENGTH NIL 8)

(DEFN HALFWORD-LENGTH NIL 16)

(DEFN WORD-LENGTH NIL 32)
```

In addition to formally characterizing the size of machine resources, constraints can be used to specify the format of resources. Consider, for instance, the format of the status register. We require that the status register contain certain flags and other fields, but we do not care which bits are assigned to these roles.

For our instruction set architecture, the status register is required to contain flags for signalling the following conditions: arithmetic overflow, address error, and request for user trap. In addition, it is required to have an 8-bit field that contains a trap code which a program uses to identify a trap.

We can state a constraint which requires that all of these fields occur within the status register, and that they all be disjoint. The constraint introduces eight new function symbols, four for accessing the fields described above, and four for constructing a new status register value given an old status register and a new field value. For the sake of brevity, we display only some of the constraints on the function symbols **OVERFLOW?** and **SET-OVERFLOW**, the accessor function and constructor function, respectively, for the overflow flag. The witnesses for these two functions assign bit 8 of the status register to be the overflow flag.

```
(CONSTRAIN
    SR-ACCESS-CONSTRAINTS NIL
    (AND (OR (TRUEP (OVERFLOW? SR))
             (FALSEP (OVERFLOW? SR)))
         (IMPLIES (BV-WORDP SR (WORD-LENGTH))
                  (BV-WORDP (SET-OVERFLOW BIT SR)
                            (WORD-LENGTH)))
         (IMPLIES (AND (BV-WORDP SR (WORD-LENGTH))
                       (EQUAL (BV-LENGTH BV) (BYTE-LENGTH)))
                  (EQUAL (OVERFLOW? (SET-OVERFLOW B SR))
                         (TRUEP B))))
    ((OVERFLOW? (LAMBDA (SR) (BV-GET 8 SR)))
     (SET-OVERFLOW (LAMBDA (BIT SR) (BV-PUT 8 BIT SR)))))
```

3.2 Constraining an Instruction

Having stated constraints on a machine state, we now turn to the problem of specifying machine instructions. We specify an ADD instruction as an example.

First, we give some functions which specify the behavior of an ALU when supplied with two bit vector arguments for addition. The function **BV-IPLUS** specifies signed addition. It returns the bit vector representation of the integer that is the sum of the integer representation of the two arguments. (The function **IPLUS** returns the signed sum of two signed numbers.) **ALU-ADD** specifies the full behavior of the ALU for an ADD. A 2-tuple is returned. The first element is the sum as defined by **BV-IPLUS**, and the second element is the overflow condition. The overflow condition occurs when the sum is not 2s-complement representable in a bit vector of a given length.

```
(DEFN BV-IPLUS
      (BV1 BV2)
      (INT-TO-BV (IPLUS (BV-TO-INT BV1)
                        (BV-TO-INT BV2))
                 (BV-LENGTH BV1)))
```

```
(DEFN ALU-ADD
 (BV1 BV2)
 (ALU-RESULT
      (BV-IPLUS BV1 BV2)
      (NOT (TC-REPRESENTABLE-INTEGER (IPLUS (BV-TO-INT BV1)
                                            (BV-TO-INT BV2))
                                     (BV-LENGTH BV1)))))
```

The remaining part of the specification of the ADD instruction states where operands occur within the machine state, and how state is updated as a result of instruction execution. Since we are modeling a RISC architecture, we consider only a register-to-register address mode.

We introduce the new function symbol **EXECUTE-ADD** to specify the ADD instruction. It takes four arguments: **S**, a machine state; **DST**, the bit-vector representation of the destination register number; **SRC1** and **SRC2**, the bit vector representations of the register numbers of the two operands. **EXECUTE-ADD** returns an updated machine state.

We introduce **EXECUTE-ADD** by constraining its behavior on each of the four components of a machine state. The constraints are as follows.

- Register File: If the addition occurs without overflow, the destination register receives the sum. If overflow occurs then the instruction is unspecified with respect to the register file. (Therefore the implementation is free to store a result, leave the destination unchanged, or take some other action.)
- Status Register: If the addition occurs without overflow, then the status register remains unchanged, otherwise the overflow bit it set.
- Program Counter: Unchanged.
- Memory: Unchanged.

In addition to the resource constraints, we require that the addition return a "good state" as specified by the constraints on the predicate **GOOD-STATE**.

The witness function for **EXECUTE-ADD** leaves the destination register unchanged on an overflow condition. In the witness, the only state change that occurs on an overflow is the setting of the overflow bit in the status register.

```
(CONSTRAIN
 EXECUTE-ADD-CONSTRAINT NIL
 (LET
     ((NEWSTATE (EXECUTE-ADD S DST SRC1 SRC2))
      (RESULT (ALU-ADD (GET (BV-TO-NAT SRC1) (REGFILE S))
                       (GET (BV-TO-NAT SRC2) (REGFILE S)))))
     (AND (IMPLIES (NOT (ALU-ERROR RESULT))
                   (EQUAL (REGFILE NEWSTATE)
                          (PUT (BV-TO-NAT DST)
                               (ALU-VALUE RESULT)
                               (REGFILE S))))
          (EQUAL (SR NEWSTATE)
                 (IF (ALU-ERROR RESULT)
                     (SET-OVERFLOW T (SR S))
                     (SR S)))
          (EQUAL (PC NEWSTATE) (PC S))
          (EQUAL (MEMORY NEWSTATE) (MEMORY S))
          (IMPLIES (AND (GOOD-STATE S)
                        (LESSP (BV-TO-NAT DST)
                               (LENGTH (REGFILE S)))
                        (LESSP (BV-TO-NAT SRC1)
                               (LENGTH (REGFILE S)))
                        (LESSP (BV-TO-NAT SRC2)
                               (LENGTH (REGFILE S))))
                   (GOOD-STATE NEWSTATE))))
 ((EXECUTE-ADD
   (LAMBDA
    (S DST SRC1 SRC2)
    (LET
     ((RESULT (ALU-ADD (GET (BV-TO-NAT SRC1) (REGFILE S))
                       (GET (BV-TO-NAT SRC2) (REGFILE S)))))
     (IF (ALU-ERROR RESULT)
         (SET-SR (SET-OVERFLOW T (SR S)) S)
         (SET-REGFILE (PUT (BV-TO-NAT DST)
                           (ALU-VALUE RESULT)
                           (REGFILE S))
                      S)))))))
```

This completes our specification example. In this section we have discussed three functions which constrain various aspects of the architecture:

- **GOOD-STATE-CONSTRAINTS** constrains the size of machine resources,
- **SR-ACCESS-CONSTRAINTS** constrains the format of the status register, and
- **EXECUTE-ADD-CONSTRAINT** constrains the ADD machine instruction.

We have used this specification style to complete a specification for a machine architecture that includes a number of additional ALU operations, load and store instructions, conditional and unconditional branch, subroutine call and return, load address, and a trap instruction. This collection of constrained functions specifies a class of architectures. The Boyer-Moore theorem prover processes these constraints so as to which guarantee the consistency of the specification.

4. Instantiating the Specification

In this section we exhibit a part of a concrete specification for an instruction set architecture, called the **X** machine for *example machine*. In the **X** machine, the size of all resources, the format of the status register, and the effect of each instruction is completely specified. We show how the specification is proved to be compliant with the constraints, and thus a member of the class of architectures defined by the constraints.

4.1 Instantiating the Machine State

We define functions which give the size of the register file and memory. We have chosen a register file length of 32, and a memory length of 2^{32}.

```
(DEFN XREGFILE-LENGTH NIL 32)

(DEFN XMEMORY-LENGTH
      NIL
      (EXP 2 (WORD-LENGTH)))
```

The predicate **GOOD-XSTATE** recognizes an acceptable **X** machine state.

```
(DEFN GOOD-XSTATE
      (S)
      (AND (STATEP S)
           (BV-WORDLISTP (REGFILE S)
                         (WORD-LENGTH))
           (EQUAL (LENGTH (REGFILE S))
                  (XREGFILE-LENGTH))
           (BV-WORDP (PC S) (WORD-LENGTH))
           (BV-WORDP (SR S) (WORD-LENGTH))
           (BV-WORDLISTP (MEMORY S)
                         (BYTE-LENGTH))
           (EQUAL (LENGTH (MEMORY S))
                  (XMEMORY-LENGTH))))
```

To prove that **GOOD-XSTATE** conforms to the constraints established by **GOOD-STATE**, we give to the Boyer-Moore theorem prover a **FUNCTIONALLY-INSTANTIATE** event. This event generates and attempts to prove the formulas necessary to establish that **GOOD-XSTATE** satisfies the constraints established for the functional variable **GOOD-STATE**. This event has the form **(FUNCTIONALLY-INSTANTIATE <name> <type> <form> <old-name> <fsubst>)**, where **<name>** is a symbol that is the name of the event, **<type>** is a directive to the theorem prover (which we will ignore), **<old-name>** is the name of a previous **CONSTRAIN** event, **<fsubst>** is a functional substitution list, and **<form>** is the constraint contained in **<old-name>** with **<fsubst>** applied. To succeed, the instantiation event must prove **<form>** as well as some other automatically generated formulas. See [Boyer, Goldschlag, Kaufmann, and Moore 89] for details.

```
(FUNCTIONALLY-INSTANTIATE
 GOOD-XSTATE-SATISFIES-CONSTRAINTS NIL
 (AND (OR (TRUEP (GOOD-XSTATE S))
          (FALSEP (GOOD-XSTATE S)))
      (IMPLIES (GOOD-XSTATE S)
               (AND (STATEP S)
                    (BV-WORDLISTP (REGFILE S)
                                  (WORD-LENGTH))
                    (EQUAL (LESSP (LENGTH (REGFILE S)) 16)
                           F)
                    (BV-WORDP (PC S) (WORD-LENGTH))
                    (BV-WORDP (SR S) (WORD-LENGTH))
                    (BV-WORDLISTP (MEMORY S)
                                  (BYTE-LENGTH))
                    (EQUAL (REMAINDER (LENGTH (MEMORY S)) 4)
                           0)))
      (IMPLIES (AND (GOOD-XSTATE STATE)
                    (STATEP S)
                    (BV-WORDLISTP (REGFILE S)
                                  (WORD-LENGTH))
                    (EQUAL (LENGTH (REGFILE S))
                           (LENGTH (REGFILE STATE)))
                    (BV-WORDP (PC S) (WORD-LENGTH))
                    (BV-WORDP (SR S) (WORD-LENGTH))
                    (BV-WORDLISTP (MEMORY S)
                                  (BYTE-LENGTH))
                    (EQUAL (LENGTH (MEMORY S))
                           (LENGTH (MEMORY STATE))))
               (GOOD-XSTATE S)))
 GOOD-STATE-CONSTRAINTS
 ((GOOD-STATE GOOD-XSTATE)))
```

In a similar event, we instantiate the functions that constrain the format of the status register. We assign specific bits within the status register to the interpretations required by the constraint. Recall from Section 3.1 that four fields are required: an overflow bit, and address error bit, a trap bit, and a trap code field. The specification of the status register leaves room for assigning additional fields in the status register. In the **X** architecture, another bit is used to flag an opcode error. We can make this assignment without affecting the correctness of the other assignments. (We do not display the instantiation event here.)

4.2 Instantiating an Instruction

In this section we display a concrete specification for the ADD instruction with the function **XEXECUTE-ADD**. This function makes explicit what happens on an ADD when an overflow occurs. In this architecture, the destination register is updated with a truncated value on overflow. (The fact that **ALU-ADD** returns a truncated value is a result of the specification function **BV-IPLUS**.)

```
(DEFN XEXECUTE-ADD
 (S DST SRC1 SRC2)
 (LET
     ((RESULT (ALU-ADD (GET (BV-TO-NAT SRC1) (REGFILE S))
                       (GET (BV-TO-NAT SRC2) (REGFILE S)))))
     (IF (ALU-ERROR RESULT)
         (SET-SR (XSET-OVERFLOW T (SR S))
                 (SET-REGFILE (PUT (BV-TO-NAT DST)
                                   (ALU-VALUE RESULT)
                                   (REGFILE S))
                              S))
         (SET-REGFILE (PUT (BV-TO-NAT DST)
                           (ALU-VALUE RESULT)
                           (REGFILE S))
                      S))))
```

The instantiation event **XEXECUTE-ADD-SATISFIES-CONSTRAINT** establishes that **XEXECUTE-ADD** satisfies the constraints established by **EXECUTE-ADD**.

```
(FUNCTIONALLY-INSTANTIATE
 XEXECUTE-ADD-SATISFIES-CONSTRAINT NIL
 (LET
     ((NEWSTATE (XEXECUTE-ADD S DST SRC1 SRC2))
      (RESULT (ALU-ADD (GET (BV-TO-NAT SRC1) (REGFILE S))
                       (GET (BV-TO-NAT SRC2) (REGFILE S)))))
     (AND (IMPLIES (NOT (ALU-ERROR RESULT))
                   (EQUAL (REGFILE NEWSTATE)
                          (PUT (BV-TO-NAT DST)
                               (ALU-VALUE RESULT)
                               (REGFILE S))))
          (EQUAL (SR NEWSTATE)
                 (IF (ALU-ERROR RESULT)
                     (XSET-OVERFLOW T (SR S))
                     (SR S)))
          (EQUAL (PC NEWSTATE) (PC S))
          (EQUAL (MEMORY NEWSTATE) (MEMORY S))
          (IMPLIES (AND (GOOD-XSTATE S)
                        (LESSP (BV-TO-NAT DST)
                               (LENGTH (REGFILE S)))
                        (LESSP (BV-TO-NAT SRC1)
                               (LENGTH (REGFILE S)))
                        (LESSP (BV-TO-NAT SRC2)
                               (LENGTH (REGFILE S))))
                   (GOOD-XSTATE NEWSTATE))))
 EXECUTE-ADD-CONSTRAINT
 ((GOOD-STATE GOOD-XSTATE)
  (EXECUTE-ADD XEXECUTE-ADD)
  (TRAPCODE XTRAPCODE)
  (OVERFLOW? XOVERFLOW?)
  (ADDRESS-ERROR? XADDRESS-ERROR?)
  (TRAPFLAG? XTRAPFLAG?)
  (SET-TRAPCODE XSET-TRAPCODE)
  (SET-OVERFLOW XSET-OVERFLOW)
  (SET-ADDRESS-ERROR XSET-ADDRESS-ERROR)
  (SET-TRAPFLAG XSET-TRAPFLAG)))
```

We have defined a complete **X** architecture, and proved that it satisfies the constraints stated for the specification architecture. The specification stops with a collection of constrained functions that describe the state and instruction set. The **X** specification is carried further to define a complete fetch-execute cycle. The function **XFETCH-EXECUTE** below takes a machine state as argument, fetches and interprets the current instruction according to a particular instruction format, and returns an updated machine state.

```
(DEFN XFETCH-EXECUTE
     (S)
     (IF (XVALID-ADDRESS (XINSTRUCTION-FETCH-ADDRESS (PC S))
                         (XFULLWORD-MODE)
                         (MEMORY S))
         (XEXECUTE-INSTRUCTION (XFETCH-INSTRUCTION S)
                               (XINCREMENT-PC S))
         (SET-SR (XSET-ADDRESS-ERROR T (SR S))
                 S)))
```

5. Summary

The use of functional instantiation makes it possible to specify a family of instruction set architectures. The elements of the specification method are:

1. Introduce a "good state" predicate to constrain the size of machine resources.
2. Introduce any necessary constraints on the format of resources, as was done with the status register above.
3. Introduce a constraint for each machine instruction. A constraint should state how an instruction modifies each of the machine resources, and should require the instruction to preserve a "good state".

The Boyer-Moore theorem prover provides mechanical support for checking the consistency of these specifications. The prover can also be used to mechanically check the proof that a particular architecture specification satisfies the constraints established by the constrained specification. Any properties proved of the constrained architecture specification are inherited by a particular architecture.

We have carried out these steps for a simple RISC architecture modeled after a MIPS machine. The specification for the concrete architecture **X** is at the same level of abstraction as Hunt's specification for FM8502 [Hunt 87]. The techniques used by Hunt to verify a gate-level implementation of FM8502 can be used to verify an implementation of the **X** machine.

Appendix A
The Boyer-Moore Logic

A complete and precise definition of the logic can be found in [Boyer & Moore 88].

We use the prefix syntax of Pure Lisp to write down terms. For example, we write **`(PLUS I J)`** where others might write *PLUS*(*I*,*J*) or *I*+*J*. We write **`(IMPLIES (P X) (EQUAL (F X) (G X)))`** in place of $P(X) \rightarrow F(X) = G(X)$.

The logic is first-order and contains no quantifiers. It is defined as an extension of propositional calculus with variables, function symbols, and the equality relation. Axioms define the following:

- the Boolean objects **`(TRUE)`** and **`(FALSE)`**, abbreviated **`T`** and **`F`**;
- The if-then-else function, **`IF`**, with the property that **`(IF X Y Z)`** is **`Z`** if **`X`** is **`F`** and **`Y`** otherwise;
- the Boolean "connector functions" **`AND`**, **`OR`**, **`NOT`**, and **`IMPLIES`**; for example, **`(NOT P)`** is **`T`** if **`P`** is **`F`** and **`F`** otherwise;
- the equality function **`EQUAL`**, with the property that **`(EQUAL X Y)`** is **`T`** or **`F`** according to whether **`X`** is **`Y`**;
- inductively constructed objects, including:
  - Natural Numbers. Natural numbers are built from the constant **`(ZERO)`** by successive applications of the constructor function **`ADD1`**. The function **`NUMBERP`** recognizes natural numbers, e.g., is **`T`** or **`F`** according to whether its argument is a natural number or not. The function **`SUB1`** returns the predecessor of a non-**`0`** natural number.
  - Ordered Pairs. Given two arbitrary objects, the function **`CONS`** returns an ordered pair containing them. The function **`LISTP`** recognizes such pairs. The functions **`CAR`** and **`CDR`** return the two components of such a pair.
- Each of the classes above is called a "shell", which can be thought of as a data type. **`T`** and **`F`** are each considered the elements of two singleton shells. Axioms insure that all shell classes are disjoint;
- the definitions of several useful functions, including:
  - **`LESSP`** which, when applied to two natural numbers, returns **`T`** or **`F`** according to whether the first is smaller than the second;
  - **`COUNT`** which, when applied to an inductively constructed object, returns its "size;" for example, the **`COUNT`** of an ordered pair is one greater than the sum of the **`COUNT`**s of the components.

The user can add of new shells, i.e., new data types. A shell defines a new class of **`n`**-tuples with type restrictions on each component. For each shell there is a recognizer (e.g., **`LISTP`** for the ordered pair shell), a constructor (e.g., **`CONS`**), an optional empty object (e.g., there is none for the ordered pairs but **`(ZERO)`** is the empty natural number), and **`n`** accessors (e.g., **`CAR`** and **`CDR`**).

The logic provides a principle of recursive definition under which new function symbols may be introduced. Consider the definition of the list concatenation function **APPEND**.

```
(DEFN APPEND
      (X Y)
      (IF (LISTP X)
          (CONS (CAR X) (APPEND (CDR X) Y))
          Y))
```

The equations submitted as definitions are accepted as new axioms under certain conditions that guarantee that one and only one function satisfies the equation. One of the conditions is that certain derived formulas be theorems. Intuitively, these formulas insure that the recursion "terminates" by exhibiting a "measure" of the arguments that decreases, in a well-founded sense, in each recursion. A suitable derived formula for **APPEND** is the following.

```
(IMPLIES (LISTP X)
         (LESSP (COUNT (CDR X)) (COUNT X)))
```

However, in general the user of the logic is permitted to choose an arbitrary measure function (**COUNT** was chosen above) and one of several relations (**LESSP** above).

The rules of inference of the logic, in addition to those of propositional calculus and equality, include mathematical induction. The formulation of the induction principle is similar to that of the definitional principle. To justify an induction schema it is necessary to prove certain theorems that establish that, under a given measure, the inductive hypotheses are about "smaller" objects than the conclusion.

Using induction it is possible to prove such theorems as the associativity of **APPEND**.

```
(PROVE-LEMMA ASSOCIATIVITY-OF-APPEND
             NIL
             (EQUAL (APPEND (APPEND A B) C)
                    (APPEND A (APPEND B C))))
```

Acknowledgements

Thanks to Bishop Brock, Matt Wilding and Bill Young for reading drafts of this paper.

References

[Boyer & Moore 88]
R. S. Boyer and J S. Moore.
A Computational Logic Handbook.
Academic Press, Boston, 1988.

[Boyer, Goldschlag, Kaufmann, and Moore 89]
R.S. Boyer, D. Goldschlag, M. Kaufmann, J S. Moore.
Functional Instantiation in First Order Logic.
Technical Report 44, Computational Logic, Inc., 1717 West Sixth Street, Suite 290 Austin, TX 78703, May, 1989.

[Firth 87] R. Firth.
Core Set of Assembly Language Instructions for MIPS-based Microprocessors.
Technical Report, Software Engineering Institute, 580 South Aiken Avenue, Pittsburgh, PA, 15213, January, 1987.

[Hennessey 82] J.L. Hennessey, N. Jouppi, J. Gill, F. Baskett, A. Strong, T. Gross, C. Rowen, J. Leonard.
The MIPS Machine.
In *IEEE Compcon*. 1982.

[Hunt 87] Warren A. Hunt, Jr.
The Mechanical Verification of a Microprocessor Design.
Technical Report CLI-6, Computational Logic, Inc., 1717 West Sixth Street, Suite 290 Austin, TX 78703, 1987.

Table of Contents