Microprocessor Design Verification

Warren A. Hunt, Jr.

Technical Report 48 September 1989

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

This report will appear in a Fall 1989 issue of the Journal of Automated Reasoning. 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 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.

Abstract

Abstract. The verification of a microprocessor design has been accomplished using a mechanical theorem prover. This microprocessor, the FM8502, is a 32-bit general-purpose, von Neumann processor whose design-level (gate-level) specification has been verified with respect to its instruction-level specification. Both specifications were written in the Boyer-Moore logic, and the proof of correctness was carried out with the Boyer-Moore theorem prover.

Chapter 1
INTRODUCTION

The verification of the FM8502 microprocessor design has been accomplished using a mechanical theorem prover. Specifications at the instruction and design level were written in the Boyer-Moore logic. A mechanization of the Boyer-Moore logic, which includes a mechanical theorem prover, was used to record the specifications and process the proof. This proof demonstrates a correlation between two models: simple Boolean logic and a much more abstract instruction-by-instruction specification. The Boolean logic design-level specification does not attempt to model physical characteristics (e.g., power requirements, clock speed) of an actual implementation; therefore, this proof does not guarantee that a physical implementation of the FM8502 wiring specification will operate. The 32-bit FM8502 is quite similar to the 16-bit FM8501 microprocessor [Hunt 85, Hunt 87]; however, the FM8502 has a richer instruction set facilitating finer control of the flag registers.[1]

The FM8502 microprocessor is a general-purpose, 32-bit, von Neumann microprocessor. Its two-address instruction format includes four addressing modes. Conditional move instructions provide program branching and each flag register can be conditionally updated on an instruction-by-instruction basis. Although there are 2^{19} FM8502 instructions, this processor is quite simple by commercial standards; the FM8502 does not contain interrupts or supervisor modes.

The instruction-level specification describes the FM8502 as an interpreter which computes the next state from the present state. The instruction-level state contains a 2^{32} word memory, an eight element register file which includes the program counter, and flag registers. The next state is computed by decoding the instruction pointed to by the program counter and then computing a new instruction-level state.

Besides providing a wiring specification for the FM8502, the design-level specification includes an interpreter which "steps" once with each micro-code instruction. The next internal FM8502 state is computed as a function of the present state. The FM8502 microprocessor itself does not contain any memory; the memory system is specified as a separate process which communicates with the FM8502 by shared connections, e.g., the memory bus. The design-level specification can be thought of as a wiring diagram composed of registers, RAM, ROM, and one- and two-input Boolean gates. The design-level specification also specifies the operation of the memory, but at an abstract level.

Proving that the design-level specification implements the instruction-level specification

---

[1]This paper describes the FM8502 on its own. However, the FM8502 is a simple perturbation of the FM8501 design except for the inclusion of a new ALU design. This paper does not contain any significant new results, it simply presents the FM8502, as this machine provides the basis for the CLI short stack [BevierHuntYoung 87]. The FM8502 was created in 1987, to provide J Moore with a more suitable target vehicle for his verified assembly language Piton.

demonstrates the correctness of the FM8502 design. The instruction-level specification computes the next programmer state as a function of the present state in one step. The design-level specification, i.e., the FM8502 design, requires many states to execute one instruction-level instruction. The FM8502 design contains a micro-coded program which interprets instructions by sequencing internal data through the appropriate internal functional units and interfacing with memory until each instruction is completed. The correctness proof of the design-level specification demonstrates that the internal functional units when controlled by the micro-code program correctly interpret instruction-level instructions. This proof involves proving the correctness of the FM8502 internal functional units, the micro-code program, and the memory interface.

The Boyer-Moore theorem prover is used to establish the correctness of the FM8502. This mechanized theorem prover, although fairly automatic, is led to the final correctness theorem by a carefully ordered set of theorem prover commands. Difficult theorems are mechanically verified by a graduated sequence of lemmas until the final result can be obtained. The FM8502 proof proceeds by axiomatizing numbers, integers, and bit vectors; defining a model for Boolean logic; composing Boolean logic elements into an ALU; proving the ALU correct; defining the instruction-level specification; defining the implementation of the FM8502, including the microcode; and, finally, proving the correctness of the design.

The correctness proof for the FM8502 does not guarantee that an implementation of the FM8502 design will result in a working microprocessor. The FM8502 design-level model is crude by hardware manufacturing standards; it models only the logical operation of simple Boolean gates and registers. The FM8502 design-level model does not model physical requirements, e.g., power, cooling, delay, etc. One might wonder whether the verification is worthwhile under these circumstances; however, just ensuring the correctness of the gate-level wiring diagram is a critical first step to predictable computer operation. Indeed, with more detailed design-level models it may be possible to verify physical resource requirements as well.

The remainder of this paper begins by introducing the concept of hardware verification through an example. Bit vectors are next axiomatized, followed by conversion functions between bit vectors and numbers. We next present the instruction-level and design-level specifications. A sketch of the FM8502 proof is given along with some proof statistics and conclusions.

Chapter 2
AN INTRODUCTION TO HARDWARE VERIFICATION

Central to the concept of hardware verification are formal demonstrations of designs meeting their more abstract specifications [LCF_LSM 81, Gordon 85, Joyce 88]. Formal logics (e.g., HOL [HOL 87], Boyer-Moore) are being adapted to the specification and verification of hardware circuits, thus enabling the use of rigorous mathematical methods, supported by mechanical theorem provers, for many aspects of circuit design. Here we introduce the use of the Boyer-Moore logic as both a circuit design language and as a specification language.

Let us consider the verification of a three input majority circuit; however, before proceeding to the verification we must define our circuit specification and our circuit design.[2] Most often circuit designers write specifications in natural language augmented with charts, graphs, and tables. For instance:

> **Specification:** A three input majority circuit has three Boolean inputs and one Boolean output. If two or three of the inputs are true then the output must be true; otherwise, the output must be false.

Designs are often drawn as schematic diagrams; one possible design for our majority circuit is shown in Figure 2-1. Circuit verification proceeds by informal argument. This type of verification suffers several drawbacks: natural language is often ambiguous, verification is often difficult or impossible, and specifications do not compose.

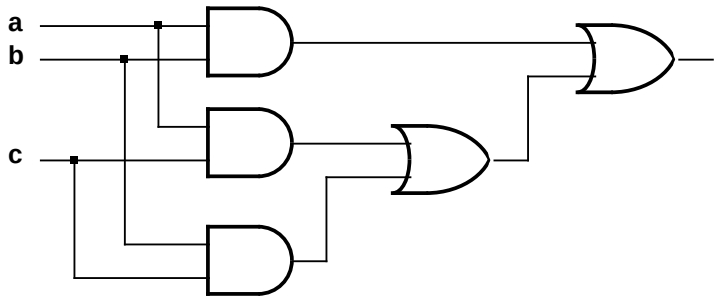

Figure 2-1: Three Input Majority Circuit

[2]Some would argue that we should attempt to refine our circuit specification into a design, while others would suggest we design our circuit directly from the specification. There are many other considerations; however, for now we proceed without concern for whether synthesis or verification is a more viable approach.

To attempt a formalization we must express ourselves with a formal language. Here we use the Boyer-Moore logic as a vehicle for expressing both specifications and designs. Specifications are often thought of as "obviously correct" abstract descriptions of intent. On the other hand, designs are concrete wiring specifications composed of Boolean gates, registers, RAMs, and ROMs. Mathematical proofs provide the verification that designs meet their specifications. The use of a formal logic allows for unambiguous specifications, rigorous verification, and abstraction composition.

The Boyer-Moore logic is an untyped, quantifier-free, first-order predicate calculus with equality that is written in a Lisp-style, prefix notation. The axiomatization of the Boyer-Moore logic defines two different constants (i.e., functions with no arguments), **(TRUE)** and **(FALSE)**, which we abbreviate with **T** and **F**, respectively. The three-argument function, **IF**, is the fundamental logical connective. **IF** is axiomatized as **(IF A B C) = C** when **A** is **F**; otherwise **(IF A B C) = B**. Other logical connectives are defined in terms of **IF**. A Peano-like framework is used to define natural numbers, and the function **PLUS** adds two natural numbers. The function **LESSP** compares two values and returns **T** if its second argument represents a natural number greater than its first argument.

We now formalize the specification of a three input majority circuit by defining the following two functions.[3]

```
(BOOL-TO-NAT X) = (IF X 1 0)

(MAJ3-SPEC A B C) = (LESSP 1 (PLUS (BOOL-TO-NAT A)
                                   (PLUS (BOOL-TO-NAT B)
                                         (BOOL-TO-NAT C))))
```

The function **BOOL-TO-NAT** converts a Boolean to **1** or **0**; this can be thought of as an abstraction function (from Booleans to natural numbers.) **MAJ3-SPEC** returns true when the sum of the three abstracted Booleans inputs is strictly greater than 1. Many designs are possible; we just formalize the schematic in Figure 2-1 by modeling the gates with functions and the wires by function composition.

```
(B-AND X Y) = (IF X (IF Y T F) F)
(B-OR  X Y) = (IF X T (IF Y T F))

(MAJ3-DESIGN A B C) = (B-OR (B-AND A B)
                            (B-OR (B-AND B C)
                                  (B-AND C A)))
```

The functions **B-AND** and **B-OR** model the operation of two Boolean gates; the function **MAJ3-DESIGN** describes the gate interconnections. Verification proceeds by exhaustive case analysis, and we state the provable relation between the specification and the design as:

```
(EQUAL (MAJ3-SPEC A B C)
       (MAJ3-DESIGN A B C))
```

This example hints at the rich character of using a general-purpose formal logic. Many hardware validation/verification procedures provide only Boolean data types. Our formal specification only concerns numbers after we apply our abstraction function **BOOL-TO-NAT**. This richness is used to great advantage in the FM8502 instruction-level specification.

---

[3]Function definitions are denoted by **=**. The left-hand side of the **=** gives the function name with its formal parameters, and the right side gives the definition of the function. Lemmas are stated without any use of the **=** symbol.

Chapter 3
BIT VECTORS

Bit vectors are used extensively in both the FM8502 instruction-level and the design-level specifications. Here we define bit vectors after introducing the Boyer-Moore logic and natural numbers.

3.1 The Boyer-Moore Logic

The Boyer-Moore logic is unusual in the regard that the logic may be extended. Most often logics are defined all at one time and are used without change for the purpose at hand. The application of any of the several *axiomatic acts* allowed by the Boyer-Moore logic extend the logic; these acts are the application of the *Shell Principle*, the *Principle of Definition*, or the addition of an arbitrary formula as an axiom. Each act *adds* a set of axioms in the means prescribed by the principle being used, or, in the case of adding an arbitrary formula, just the singleton set consisting of the arbitrary formula. A *history* is just a finite sequence of axiomatic acts. The order of the history is important; for instance, it is not possible to use bit vectors before they are defined. Adding an arbitrary formula as an axiom does not guarantee the soundness of the logic.[4]

A theorem is any axiom or something that can be reduced to axioms by using rules of inference. Rules of inference include: modus ponens, deduction, case analysis, induction, and substitution of equals for equals. Any instance of a theorem is a theorem, and any propositional tautology is a theorem.

The Boyer-Moore logic comes pre-packaged with data types for Booleans, literal atoms, pairs, natural numbers, and the negative integers. Also included are many definitions operating on these data types, which we present as necessary to support our presentation. Axioms arising from defining the built-in data types can be found elsewhere [Boyer & Moore 88], but we do present the axioms generated as a result of using the Shell Principle to define bit-vectors.

[4]The definition of a formal logic is a subtle thing; new definitions and data types must be introduced in such a way that no inconsistencies manifest themselves. We ignore this subject here and take as given the soundness and consistency of the Boyer-Moore logic and its associated theorem-proving system.

3.2 Natural Numbers

Before attempting to define bit vectors, we quickly present some definitions about natural numbers. Natural numbers are axiomatized in a Peano-like fashion. We abbreviate natural numbers (and integers) with (signed) numerals, thus we abbreviate **(ADD1 (ZERO))** with **1**. Built into the Boyer-Moore logic are the following arithmetic functions.

```
(ZEROP X) = (OR (EQUAL X (ZERO)) (NOT (NUMBERP X)))

(FIX X) = (IF (NUMBERP X) X 0)

(PLUS X Y) = (IF (ZEROP X)
                 (FIX Y)
                 (ADD1 (PLUS (SUB1 X) Y)))

(LESSP X Y) = (IF (ZEROP Y)
                  F
                  (IF (ZEROP X)
                      T
                      (LESSP (SUB1 X) (SUB1 Y))))

(DIFFERENCE I J) = (IF (ZEROP I)
                       0
                       (IF (ZEROP J)
                           I
                           (DIFFERENCE (SUB1 I) (SUB1 J))))

(TIMES I J) = (IF (ZEROP I)
                  0
                  (PLUS J (TIMES (SUB1 I) J)))

(QUOTIENT I J) = (IF (ZEROP J)
                     0
                     (IF (LESSP I J)
                         0
                         (ADD1 (QUOTIENT (DIFFERENCE I J) J))))

(REMAINDER I J) = (IF (ZEROP J)
                      (FIX I)
                      (IF (LESSP I J)
                          (FIX I)
                          (REMAINDER (DIFFERENCE I J) J)))
```

3.3 The Axiomatization of Bit Vectors

Bit vectors are fundamental to the FM8502 specifications; they are used in both the design-level and the instruction-level specifications to describe the FM8502 state space. The purpose of defining bit vectors is to allow us to name a group of bits with one name, e.g., the data bus. Informally, we define a bit vector to be a proper list containing only Boolean values. We employ the Boyer-Moore Shell Principle to formally define bit vectors.

Shell Definition.
Add the shell **BITV** of two arguments, with
base function **BTM**;
recognizer function **BITVP**;
accessor functions **BIT** and **VEC**;
type restrictions **(ONE-OF FALSEP TRUEP)** and **(ONE-OF BITVP)**; and
default functions **FALSE** and **BTM**.

The axioms added as a result of this invocation of the Shell Principle are below. The first axiom says the

recognizer function **BITVP** returns a Boolean; the second axiom says the recognizer function recognizes the constructor function **BITV** with any two arguments; the third axiom says the recognizer function recognizes the base function **BTM**; and so on.

```
(OR (EQUAL (BITVP X) T)
    (EQUAL (BITVP X) F))

(BITVP (BITV X Y))

(BITVP (BTM))

(NOT (EQUAL (BITV X Y) (BTM)))

(IMPLIES (AND (BITVP X)
              (NOT (EQUAL X (BTM))))
         (EQUAL (BITV (BIT X) (VEC X)) X))

(IMPLIES (OR (FALSEP X) (TRUEP X))
         (EQUAL (BIT (BITV X Y)) X))

(IMPLIES (BITVP X)
         (EQUAL (VEC (BITV X Y) Y)))

(IMPLIES (OR (NOT (BITVP X))
             (OR (EQUAL X (BTM))
                 (AND (NOT (OR (FALSEP X1) (TRUEP X1)))
                      (EQUAL X (BITV X1 X2)))))
         (EQUAL (BIT X) F))

(IMPLIES (OR (NOT (BITVP X))
             (OR (EQUAL X (BTM))
                 (AND (NOT (BITVP X2))
                      (EQUAL X (BITV X1 X2)))))
         (EQUAL (VEC X) (BTM)))

(NOT (BITVP F))

(NOT (BITVP T))
```

And for each recognizer function, **recognizer**, defined in the present history, the following is an axiom.[5]

```
(IMPLIES (BITVP X) (NOT (recognizer X)))
```

3.4 Bit Vector Concatenation

After this invocation of the Shell Principle, the logic (the present history) can be extended by function definitions which refer to the bit vector data type. We employ the Principle of Definition to define a recognizer for empty bit vectors; we consider other data types to be empty insofar as bit vectors are concerned.

```
(BTMP X) = (IF (BITVP X) (EQUAL X (BTM)) T)
```

Proper application of the Principle of Definition ensures that a new function terminates. The termination of non-recursive functions is trivial; however, in the case of recursively defined functions, termination is demonstrated by showing some well-founded measure decreases with each recursive call. The recursive function **V-APPEND** appends two bit vectors together.

---

[5]Several other axioms are introduced which we do not make reference to.

```
(V-APPEND X Y) = (IF (BTMP X)
                     Y
                     (BITV (BIT X)
                           (V-APPEND (VEC X) Y)))
```

V-APPEND has only one recursive call which is guarded by **(NOT (BTMP X))**; in this case **X** must be a non-empty bit vector ensuring that **(VEC X)** is smaller than **X** in some well-founded sense.[6]

3.5 Bit Vector Functions

Any object can be coerced into a bit vector by the function **BV-FIX**. To measure the length of a bit vector we use the function **SIZE**. The function **TRUNC** is used to truncate (or extend) a bit vector to a specific length. **BITN** selects a specific bit from a vector; bits are numbered starting at 1.

```
(BV-FIX X) = (IF (BITVP X) X (BTM)))

(SIZE A) = (IF (BTMP A)
               0
               (ADD1 (SIZE (VEC A))))

(TRUNC A N) = (IF (ZEROP N)
                  (BTM)
                  (BITV (BIT A)
                        (TRUNC (VEC A) (SUB1 N))))

(BITN X N) = (IF (ZEROP N)
                 F
                 (IF (EQUAL N 1)
                     (BIT X)
                     (BITN (VEC X) (SUB1 N)))))

(COMPL X) = (IF (BTMP X)
                (BTM)
                (BITV (NOT (BIT X))
                      (COMPL (VEC X)))))

(INCR C X) = (IF (BTMP X)
                 (BTM)
                 (BITV (XOR C (BIT X))
                       (INCR (AND C (BIT X)) (VEC X)))))
```

COMPL complements a bit vector, and **INCR** is a ripple carry increment function. We often use **SIZE** in proofs about inductively defined hardware functions of more than one argument.

Once we have defined these functions, we may wish to prove certain properties about them. For instance, we can prove by induction that **TRUNC** returns a vector **N** bits long for all natural numbers **N**. We write such a lemma as follows.

```
(IMPLIES (NUMBERP N)
         (EQUAL (SIZE (TRUNC A N)) N)))
```

[6]For a complete introduction to the Principle of Definition, see Boyer and Moore's book [Boyer & Moore 88].

3.6 Theorem Prover Usage

The mechanization of the FM8502 is interesting in its own right. Each of the proofs discussed here could be carried out by hand (assuming one had enough time), but the task of carrying out the entire proof by hand would be nearly impossible. The Boyer-Moore theorem prover removes much of the tedium involved in proving; it is used as a theorem-proving assistant. This assistant manages the evolving history of events (definitions, prove commands, etc.), thus relieving the user from many mental chores. But the importance of the user in guiding the system cannot be overemphasized.

The concrete syntax for the Boyer-Moore theorem prover proof command **PROVE-LEMMA** requires that a name and "lemma class" be included as well as the theorem to be proved. The "lemma class" specifies how the lemma is to be used as a rule. Upon the successful completion of a **PROVE-LEMMA** command, the theorem prover incorporates the theorem into its database. Below is an actual invocation of the Boyer-Moore theorem prover on the above lemma, with this event named **TRUNC-HAS-CORRECT-SIZE** and its class being **REWRITE**. The Boyer-Moore theorem prover requires 0.7 seconds to create and print the proof below. The lemma generates a rewrite rule that causes terms of the form **(SIZE (TRUNC A N))** to rewrite to **N**, wherever **N** can be shown to be numeric.

```
(PROVE-LEMMA TRUNC-HAS-CORRECT-SIZE
             (REWRITE)
             (IMPLIES (NUMBERP N)
                      (EQUAL (SIZE (TRUNC A N)) N)))

     Name the conjecture *1.

     We will appeal to induction.  There is only one plausible
induction.  We will induct according to the following scheme:
      (AND (IMPLIES (ZEROP N) (p A N))
           (IMPLIES (AND (NOT (ZEROP N))
                         (p (VEC A) (SUB1 N)))
                    (p A N))).
Linear arithmetic, the lemma COUNT-NUMBERP, and the definition of
ZEROP inform us that the measure (COUNT N) decreases according to
the well-founded relation LESSP in each induction step of the
scheme.  Note, however, the inductive instance chosen for A.  The
above induction scheme produces the following two new formulas:

Case 2. (IMPLIES (AND (ZEROP N) (NUMBERP N))
                 (EQUAL (SIZE (TRUNC A N)) N)).

  This simplifies, expanding the definitions of ZEROP, NUMBERP,
  EQUAL, TRUNC, and SIZE, to:

        T.
```

```
Case 1. (IMPLIES (AND (NOT (ZEROP N))
                      (EQUAL (SIZE (TRUNC (VEC A) (SUB1 N)))
                             (SUB1 N))
                      (NUMBERP N))
                 (EQUAL (SIZE (TRUNC A N)) N)).

  This simplifies, applying ADD1-SUB1 and VEC-BITV, and expanding
  the definitions of ZEROP, TRUNC, BTMP, and SIZE, to:

        T.


     That finishes the proof of *1.  Q.E.D.


[ 0.0 0.4 0.3 ]
```

3.7 Implicit Assumptions

The definitions and data type presented above are offered as "obviously correct." Many other (equally good) ways of defining bit vectors are possible, including our bit vector functions. The usefulness of verification is best demonstrated when top-level specifications are simple and abstract. Proofs cannot put meaning into specifications. Designs proven to implement meaningless specifications are also meaningless [Cohn 89].

The many valid questions concerning the usefulness of verification we leave unanswered; however, we believe it is faster to verify a microprocessor design than to exhaustively test one.

Chapter 4
BIT VECTOR ABSTRACTION FUNCTIONS

The FM8502 instruction-level specification involves both time and data abstractions from the design-level specification. In this section we are just concerned with data abstractions which allow us to view bit-vectors as natural numbers and integers. This section continues presenting "obviously correct" functions; that is, these functions are used in the instruction-level specification.

4.1 Natural Numbers

The natural number interpretation of a bit-vector is defined by the function **BV-TO-NAT**; its inverse is named **NAT-TO-BV**. Our natural number interpretation of bit vectors is "little endian", i.e., the least significant bit is first. We use the function **CARRY** for converting a single bit to **1** or **0**.

```
(BV-TO-NAT X) = (IF (BTMP X)
                    0
                    (PLUS (IF (BIT X) 1 0)
                          (TIMES 2 (BV-TO-NAT (VEC X)))))

(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))))

(CARRY C) = (IF C 1 0)
```

Using **NAT-TO-BV** for **N** greater than can be represented in **SIZE** bits causes truncation; however, the following is a theorem.

```
(IMPLIES (BITVP X)
         (EQUAL (NAT-TO-BV (BV-TO-NAT X) (SIZE X)) X))
```

4.2 Integers

Integers are represented as signed natural numbers: positive integers are represented as natural numbers, negative numbers are written as **(MINUS N)** and abbreviated **-N**. The function **MINUS** is a shell constructor with recognizer function **NEGATIVEP** and accessor function **NEGATIVE-GUTS**.[7]

We recognize integers with the function **INTEGERP**, and function **INTEGER-FIX** coerces an

[7]This definition of integers admits **(MINUS 0)**; however, our "good" integer predicate, **INTEGERP**, does not recognize **(MINUS 0)**.

argument to be an integer. **INTEGER-MINUS** negates an integer.

```
(INTEGERP X) =  (OR (NUMBERP X)
                    (AND (NEGATIVEP X)
                         (NOT (ZEROP (NEGATIVE-GUTS X)))))

(INTEGER-FIX X) = (IF (INTEGERP X) X 0)

(INTEGER-MINUS X) = (IF (NEGATIVEP X)
                        (NEGATIVE-GUTS X)
                        (IF (ZEROP X) 0 (MINUS X)))
```

Our integer addition function **ADD** is defined in terms of natural number addition and subtraction. Integer subtraction is defined in terms of integer addition.

```
(ADD X Y) = (IF (NEGATIVEP X)
                (IF (NEGATIVEP Y)
                    (MINUS (PLUS (NEGATIVE-GUTS X) (NEGATIVE-GUTS Y)))
                    (IF (LESSP Y (NEGATIVE-GUTS X))
                        (MINUS (DIFFERENCE (NEGATIVE-GUTS X) Y))
                        (DIFFERENCE Y (NEGATIVE-GUTS X))))
                (IF (NEGATIVEP Y)
                    (IF (LESSP X (NEGATIVE-GUTS Y))
                        (MINUS (DIFFERENCE (NEGATIVE-GUTS Y) X))
                        (DIFFERENCE X (NEGATIVE-GUTS Y)))
                    (PLUS X Y))))

(SUB X Y) = (ADD X (INTEGER-MINUS Y))
```

Integer addition is complicated by two facts: two data types are used to represent integers; and **(DIFFERENCE 5 3)** equals **2**, but **(DIFFERENCE 3 5)** equals **0**. Therefore, we must identify both the sign and magnitude of the input integers to ensure that the first argument to **DIFFERENCE** is greater than or equal to the second.

It is unfortunate that the definition of integer addition is so complicated, as this is another "obviously correct" function. To enhance our belief in the "correctness" of the definition of **ADD**, we prove a number of theorems about **ADD**, such as its commutivity and associativity.

We use a two's complement representation for integers. The function **INTEGER-IN-RANGEP** checks to see if integer **N** can be represented without truncation in **SIZE** bits. Below, **BV-TO-INT** converts bit vectors into integers and **INT-TO-BV** converts an integer into a bit vector with **SIZE** bits.

```
(INTEGER-IN-RANGEP N SIZE) = (IF (ZEROP SIZE)
                                 F
                                 (IF (NEGATIVEP N)
                                     (NOT (LESSP (EXP 2 (SUB1 SIZE))
                                                 (NEGATIVE-GUTS N)))
                                     (LESSP N (EXP 2 (SUB1 SIZE))))))

(INT-TO-BV X SIZE) = (IF (NEGATIVEP X)
                         (INCR T (COMPL (NAT-TO-BV (NEGATIVE-GUTS X)
                                                   SIZE)))
                         (NAT-TO-BV X SIZE)))

(BV-TO-INT X) = (IF (BITN X (SIZE X))
                    (MINUS (BV-TO-NAT (INCR T (COMPL X))))
                    (BV-TO-NAT X)))
```

We also define functions for converting natural numbers to integers and back again directly. These functions can be thought of as first converting their input number into a bit vector with **SIZE** bits and then on to an integer or natural number as appropriate.

```
(NAT-TO-INT N SIZE) = (IF (LESSP N (EXP 2 (SUB1 SIZE)))
                          N
                          (MINUS (DIFFERENCE (EXP 2 SIZE) N))))

(INT-TO-NAT N SIZE) = (IF (NEGATIVEP N)
                          (DIFFERENCE (EXP 2 SIZE) (NEGATIVE-GUTS N))
                          N))
```

4.3 Integer vs. Natural Number Addition

The most interesting lemma about integer addition concerns its relationship to truncating natural number addition. The FM8502 microprocessor represents both natural numbers and integers as bit vectors. To find out what natural number a particular bit vector represents, we use **BV-TO-NAT**, and for integers we use **BV-TO-INT**. Using the two's complement representation we chose for integers allows the same addition algorithms (hardware) to be used for both integer and natural number addition.

Consider any two integers **X** and **Y** which can be represented as bit vectors with **N** bits. Then it is possible to add them together by first converting them into natural numbers, adding them with **PLUS** modulo $2^{\mathbf{N}}$, and converting this result back into an integer.[8] In other words, the integer result returned by truncating natural number addition is exactly the same result returned by our **ADD** function given above. Below we state a slightly more general result which includes a carry input bit **C**, and includes the behavior of this type of addition when the result is too large to be represented with **N** bits.

```
(IMPLIES
 (AND (INTEGERP X)
      (INTEGERP Y)
      (INTEGER-IN-RANGEP X N)
      (INTEGER-IN-RANGEP Y N))
 (EQUAL (NAT-TO-INT (REMAINDER (PLUS (CARRY C)
                                     (INT-TO-NAT X N)
                                     (INT-TO-NAT Y N))
                               (EXP 2 N))
                    N)
        (IF (INTEGER-IN-RANGEP (ADD X (ADD Y (CARRY C))) N)
            (ADD X (ADD Y (CARRY C)))
            (IF (NEGATIVEP (ADD X (ADD Y (CARRY C))))
                (ADD X (ADD Y (ADD (CARRY C) (EXP 2 N))))
                (ADD X (ADD Y (ADD (CARRY C) (MINUS (EXP 2 N)))))))))
```

Integer addition wraps around in a way similar to the wrap-around behavior of truncating natural number addition.

[8] This type of truncating natural number addition is provided by hardware adders; we later demonstrate this property for our ripple carry adder **BV-ADDER**.

Chapter 5
THE FM8502 INSTRUCTION-LEVEL SPECIFICATION

The FM8502 instruction-level specification is two part: the instruction interpreter and the ALU interpretation lemmas. The instruction interpreter takes a machine state and produces a new machine state as a result of executing one instruction. This interpreter is the "programmer's reference guide"; that is, for every possible instruction and every possible machine state, this interpreter describes the operation of the FM8502. The ALU interpretation lemmas allow the results computed by the instruction interpreter to be viewed as bit-vectors, natural numbers, or integers. Before giving the formal instruction-level specification, we give an informal description of the FM8502.

5.1 An Informal Programmer's Description

The FM8502 is a conventional von Neumann processor. The FM8502 has eight general purpose registers; register zero is overloaded and acts as the program counter. There are four flag registers: carry, overflow, zero, and negative. The FM8502 can address 2^{32} words of memory. We call this state the programmer-visible state, as this is the only state an FM8502 programmer can influence directly (Figure 5-1.)

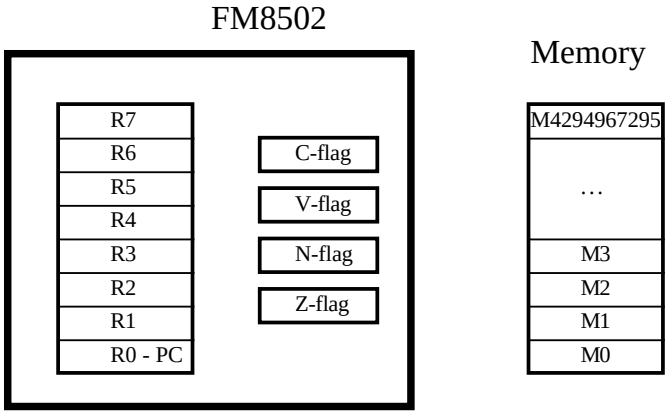

Figure 5-1: FM8502 Programmer-visible State

The FM8502 has a two address instruction format. Every instruction has one of four forms: **B←A op B**, **B←op A**, **B←(condition)A**, or **B←A**, where ← is an assignment statement and **op** is the operation to be performed. The third instruction format is a conditional move operation and the last is an unconditional move. The FM8502 instruction set is summarized in Table 5-1.

OP-CODE	MOVE	Operation	Description
0000	0	$b \leftarrow a$	Move
0001	0	$b \leftarrow a+1$	Increment
0010	0	$b \leftarrow b+a+c$	Add with carry
0011	0	$b \leftarrow b+a$	Add
0100	0	$b \leftarrow 0-a$	Negation
0101	0	$b \leftarrow a-1$	Decrement
0110	0	$b \leftarrow b-a-c$	Subtract with borrow
0111	0	$b \leftarrow b-a$	Subtract
1000	0	$b \leftarrow a>>1$	Rotate right, shifted through carry
1001	0	$b \leftarrow a>>1$	Arithmetic shift right, top bit duplicated
1010	0	$b \leftarrow a>>1$	Logical shift right, top bit zero
1011	0	$b \leftarrow b \veebar a$	Exclusive or
1100	0	$b \leftarrow b \vee a$	Or
1101	0	$b \leftarrow b \wedge a$	And
1110	0	$b \leftarrow \neg a$	Not
1111	0	$b \leftarrow a$	Move
0000	1	$b \leftarrow (\sim c)a$	Move if no carry
0001	1	$b \leftarrow (c)a$	Move if carry
0010	1	$b \leftarrow (\sim v)a$	Move if no overflow
0011	1	$b \leftarrow (v)a$	Move if overflow
0100	1	$b \leftarrow (\sim z)a$	Move if not zero
0101	1	$b \leftarrow (z)a$	Move if zero
0110	1	$b \leftarrow (\sim n)a$	Move if not negative
0111	1	$b \leftarrow (n)a$	Move if negative
1000	1	$b \leftarrow a$	Move
...			
1111	1	$b \leftarrow a$	Move

Table 5-1: FM8502 Instruction Set Summary

Table 5-1 is somewhat simplified as the flag registers may also have new values assigned. The FM8502 is unusual in allowing the programmer the option of conditionally setting each flag register.[9] Each operand, **a** and **b**, may use one of four addressing modes: register direct, register indirect, register indirect with pre-decrement, and register indirect with post-increment.

FM8502 instructions are interpreted as bit vectors; that is, when a 32-bit instruction is fetched from memory it is split in smaller bit fields and these smaller bit fields indicate what registers to use, what ALU operation to perform, etc. The diagram in Figure 5-2 shows how bit fields are arranged in an instruction.

REGA and **REGB** specify the registers for operand **A** and **B**, respectively. **MODEA** and **MODEB** specify which of the four addressing modes are to be used. **SET-C**, **SET-V**, **SET-N**, and **SET-Z** specify which flag registers are to be updated. When the **MOVE** bit is set, the type of the move instruction is specified in the **OP** field. When **MOVE** is not set, **OP** specifies an ALU operation to be applied.

Instructions are executed in six steps: the instruction pointed by the program counter is fetched from memory and the program counter is incremented; operand **A** is fetched with pre-decrement if coded; operand **B** is fetched with pre-decrement if coded; the results are computed and stored; operand **A** is

[9]The FM8501 allowed the conditional update of all the flags at once. During the verification of Piton assembler, J Moore requested the ability to conditionally set each flag. The problem he encountered was not being able to predict what the overflow flag would be when adding natural numbers, or equivalently not being able to predict what the carry flag would be when adding two integers. The ability to conditionally set each flag allowed Moore to just set the flags he was interested in and have the rest remain constant.

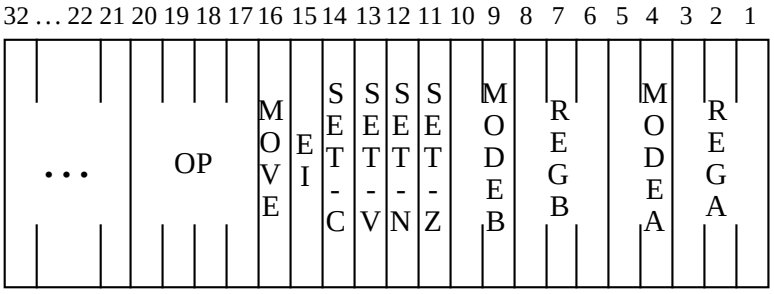

Figure 5-2: FM8502 Instruction Word Layout

post-incremented if coded; and operand **B** is post-incremented if coded.

5.2 An Introduction to the Instruction-level Specification

The need for a two-part instruction-level specification is due to the self-modifying nature of von Neumann machines. Users of the FM8502 may write data to the FM8502 memory and then execute this data. Because of this it is necessary for all the memory elements to have a uniform type. The first part of the instruction-level specification is an interpreter that steps once with each instruction executed. The second part is a set of interpretation lemmas that allow the results of the interpreter to be interpreted as bit vectors, natural numbers, or integers.

We think of the instruction-level specification as an interpreter containing an ALU. A new state is computed with the help of the ALU function **BV-ALU-CV**. This ALU takes two bit vectors, a carry bit, and an opcode as input and produces a bit vector result with a carry and an overflow. We use the interpretation lemmas to observe the workings of the instruction-level specification; that is, we can replace occurrences of **BV-ALU-CV** with one of the three interpretation lemmas. For example, imagine we want to interpret an add instruction.

```
(FM8502 PROGRAMMER-STATE time)
 =
(IF (out-of TIME)
    PROGRAMMER-STATE
    (FM8502 (compute-next-state ... (BV-ALU-CV a b c add-op-code) ...)
            (take-away-one-step-of TIME)))
```

BV-ALU-CV returns three values collected into one object by the shell constructor **BV-CV**; this constructor has accessors **BV**, **C**, and **V** for the bit vector, carry, and overflow outputs, respectively. The bit vector returned by **BV-ALU-CV** is stored as the result of the add instruction along with the carry and overflow. To examine the effect of the add instruction we first need to decide whether integers or natural numbers are being added. Let us assume we are adding natural numbers. Then we can use the natural number interpretation lemma to see the effect. The natural number interpretation lemma has the following form.

```
(AND ...
     (EQUAL (BV (BV-ALU-CV a b c add-op-code))
            (NAT-TO-BV (REMAINDER (PLUS (BV-TO-NAT a)
                                        (PLUS (BV-TO-NAT b)
                                              (CARRY c)))
                                  (EXP 2 (SIZE a)))
                       (SIZE a)))
     ...)
```

Thus we can determine what the result computed by the ALU function means if we consider our bit vector data to be representing natural numbers. The rest of this section presents the interpreter and the ALU interpretation lemmas in more detail.

5.3 The Formal Instruction-level Specification

The formal presentation of the instruction-level specification proceeds by defining the programmer-visible state, defining the instruction interpreter and then defining the ALU interpretation lemmas.

Using the Shell Principle we define a new constructor function for the programmer-visible state.

Shell Definition.
Add the shell **M-STATE** of two arguments, with
recognizer function **M-STATEP**;
accessor functions **M-REGS**, **M-C-FLG**, **M-V-FLG**, **M-Z-FLG**, **M-N-FLG**, and **M-MEM**;
type restrictions **(NONE-OF)**, **(NONE-OF)**, **(NONE-OF)**,
(NONE-OF), **(NONE-OF)**, and **(NONE-OF)**; and
default functions **ZERO**, **ZERO**, **ZERO**, **ZERO**, **ZERO**, and **ZERO**.

This six-place constructor packages the register file, the flags, and the memory into one object.

We use the function **FM8502-M-STATEP** to recognize a well-formed programmer-visible state.

```
(EVERY-MEMBER-SIZEP LST N)
 =
(IF (NLISTP LST)
    T
    (AND (EQUAL (SIZE (CAR LST)) N)
         (EVERY-MEMBER-SIZEP (CDR LST) N)))

(RAMP RAM WIDTH LOCATIONS)
 =
(AND (EQUAL (LENGTH RAM) LOCATIONS)
     (EVERY-MEMBER-SIZEP RAM WIDTH))

(FM8502-M-STATEP STATE)
 =
(LET ((WORD-SIZE 32)
      (NUM-REGS   8))
  (AND (M-STATEP STATE)
       (RAMP (M-REGS STATE) WORD-SIZE NUM-REGS)
       (RAMP (M-MEM  STATE) WORD-SIZE (EXP 2 WORD-SIZE))
       (BOOLP (M-C-FLG STATE))
       (BOOLP (M-V-FLG STATE))
       (BOOLP (M-N-FLG STATE))
       (BOOLP (M-Z-FLG STATE))))
```

We represent a random access memory as a list of bit vectors. The function **RAMP** checks such a list for suitable structure.

Before we can present the instruction-level interpreter we need to define some helper functions. To access bit fields out of a 32-bit instruction we have to define several field accessors: **BV-OP-CODE** gets a four-bit opcode; **B-MOVE-OP** gets the move bit; **B-C-SET** specifies whether the carry flag is to be set with the carry output from the ALU; **B-V-SET**, **B-N-SET**, and **B-Z-SET**, specify whether the overflow, negative, and zero flags are to be set; and **BV-OPRD-B** and **BV-OPRD-A** specify which registers are to be used to compute the values for operand **B** and operand **A**. We do not include the decoding for the addressing modes as we do not include enough detail here to see how they are used.

```
(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)))))

(B-MOVE-OP I-REG) = (BITN I-REG 16)

(B-C-SET I-REG) = (BITN I-REG 14)
(B-V-SET I-REG) = (BITN I-REG 13)
(B-N-SET I-REG) = (BITN I-REG 12)
(B-Z-SET I-REG) = (BITN I-REG 11)

(BV-OPRD-B I-REG) = (BITV (BITN I-REG 6)
                          (BITV (BITN I-REG 7)
                                (BITV (BITN I-REG 8)
                                      (BTM))))

(BV-OPRD-A I-REG) = (BITV (BITN I-REG 1)
                          (BITV (BITN I-REG 2)
                                (BITV (BITN I-REG 3)
                                      (BTM))))
```

Another helper function we need is the function which computes the result of the ALU. The data type returned by **BV-ALU-CV-RESULTS** is constructed by the shell constructor **BV-CV**.

Shell Definition.
Add the shell **BV-CV** of three arguments, with
recognizer function **BV-CVP**;
accessor functions **BV**, **C**, and **V**;
type restrictions **(ONE-OF BITVP)**, **(ONE-OF FALSEP TRUEP)**,
and **(ONE-OF FALSEP TRUEP)**; and
default functions **BTM**, **FALSE**, and **FALSE**.

The function **BV-ALU-CV** is the definition of the actual hardware ALU; the identical function appears in the design-level specification. **BV-ALU-CV** takes as operands two bit vectors, a one-bit carry, and an opcode decoded from the opcode in the current instruction. The function **BV-ALU-OP-CODE** either returns the ALU opcode for a move if the move bit is set or the op-code from the current instruction.

```
(BV-ALU-OP-CODE I-REG) = (BV-IF (B-MOVE-OP I-REG)
                                (NAT-TO-BV 0 4)
                                (BV-OP-CODE I-REG))

(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 MEMORY)))
```

The function **CURRENT-INSTRUCTION** supplies the current instruction. The functions **A-VALUE-FOR-ALU-AFTER-OPRD-A-PRE-DECREMENT** and **B-VALUE-FOR-ALU-AFTER-OPRD-B-PRE-DECREMENT** supply the two operands to the ALU after taking into account the increment of the program counter and the possible pre-decrement operations. The second part of the instruction-level specification involves a set of three lemmas which describe the operation of the ALU in terms of bit vectors, natural numbers, and integers. We return to this subject shortly.

We are now ready to present the instruction-level interpreter. This interpreter "ticks" once per instruction. A new state is computed from the present state by function **FM8502-STEP** and function

FM8502 allows a specific number of instructions to be executed.

```
(FM8502-STEP STATE)
 =
(LET ((REG-FILE (M-REGS  STATE))
      (REAL-MEM (M-MEM   STATE))
      (C-FLAG   (M-C-FLG STATE))
      (V-FLAG   (M-V-FLG STATE))
      (N-FLAG   (M-N-FLG STATE))
      (Z-FLAG   (M-Z-FLG STATE)))
  (LET ((INSTRUCTION (CURRENT-INSTRUCTION REG-FILE REAL-MEM))
        (ALU-RESULTS (BV-ALU-CV-RESULTS REG-FILE REAL-MEM C-FLAG)))
    (M-STATE (REG-FILE-AFTER-OPRD-B-POST-INCREMENT
              REG-FILE REAL-MEM C-FLAG V-FLAG Z-FLAG N-FLAG)
             (IF (B-C-SET INSTRUCTION) (C ALU-RESULTS) C-FLAG)
             (IF (B-V-SET INSTRUCTION) (V ALU-RESULTS) V-FLAG)
             (IF (B-N-SET INSTRUCTION)
                 (NEGATIVEP (BV-TO-INT (BV ALU-RESULTS)))
                 N-FLAG)
             (IF (B-Z-SET INSTRUCTION)
                 (ZEROP (BV-TO-NAT (BV ALU-RESULTS)))
                 Z-FLAG)
             (REAL-MEM-AFTER-ALU-WRITE
              REG-FILE REAL-MEM C-FLAG V-FLAG Z-FLAG N-FLAG))))

(FM8502 STATE N) = (IF (ZEROP N)
                       STATE
                       (FM8502 (FM8502-STEP STATE) (SUB1 N)))
```

The instruction-level specification of the flags is straightforward. Each flag, depending on the respective flag set bit, is either updated as a function of the ALU or remains unchanged. The carry flag is set to the carry out of the ALU, and the overflow flag is set to the overflow output of the ALU. The negative flag is set if the integer interpretation of the bit vector output of the ALU is negative. The zero flag is set if the natural number interpretation of the bit vector output of the ALU is zero. Elsewhere we prove the bit vector representation for integer zero is the same as the representation for the natural number zero.

The function **REAL-MEM-AFTER-ALU-WRITE** either updates the memory addressed by operand **B** with the bit vector output of the ALU or leaves the memory unchanged. The function **REG-FILE-AFTER-OPRD-B-POST-INCREMENT** is actually a composition of six functions, which specify the order and content of the updates which can occur to the register file. We do not present these six functions but we describe their effect. The first function increments the program counter, the second decrements the register for operand **A** if specified, the third decrements the register for operand **B** if specified, the forth updates the register file with the bit vector result of the ALU, and the fifth and sixth perform post-increment operations for operands **A** and **B** if specified.

The remaining part of the instruction-level specification is concerned with the ALU interpretation lemmas. Instead of presenting all of the ALU lemmas for bit vectors, natural numbers, and integers, we just present the natural number and integer interpretation of logical shift right (ALU op-code 10), arithmetic shift right (ALU op-code 9), and increment (ALU op-code 1). We first present the natural number interpretation of these three ALU operations. Notice that this (abbreviated) lemma gives no interpretation for the arithmetic shift right operation, as this has no clear meaning for natural numbers. Similarly, the natural number interpretation lemma gives no interpretation to the overflow output of the ALU.[10]

[10]For natural number operations, a carry output being set means the bit vector result of the ALU is incorrect; likewise, an overflow output being set means the bit vector result of the ALU is not the exact integer result.

```
(AND (EQUAL (BV-ALU-CV A B C (NAT-TO-BV 10 4))
            (BV-CV (NAT-TO-BV (QUOTIENT (BV-TO-NAT A) 2) (SIZE A))
                   (NOT (ZEROP (REMAINDER (BV-TO-NAT A) 2)))
                   F))

     (EQUAL (BV (BV-ALU-CV A B C (NAT-TO-BV 1 4)))
            (IF (LESSP (ADD1 (BV-TO-NAT A)) (EXP 2 (SIZE A)))
                (NAT-TO-BV (ADD1 (BV-TO-NAT A)) (SIZE A))
                (NAT-TO-BV 0 (SIZE A))))

     (EQUAL (C (BV-ALU-CV A B C (NAT-TO-BV 1 4)))
            (NOT (LESSP (ADD1 (BV-TO-NAT A)) (EXP 2 (SIZE A)))))
     ...)
```

The integer ALU interpretation lemmas for the same three ALU operations are given below along with the definition of the function **MOD2**. We have no interpretation for the logical shift right operation, as it has no meaning with respect to integers. The integer interpretation of the arithmetic shift right is given along with the integer interpretation of the increment operation.

```
(MOD2 X) = (IF (NEGATIVEP X)
               (MINUS (QUOTIENT (ADD1 (NEGATIVE-GUTS X)) 2))
               (QUOTIENT X 2))


(AND (EQUAL (BV (BV-ALU-CV A B C (NAT-TO-BV 9 4)))
            (INT-TO-BV (MOD2 (BV-TO-INT A))
                       (SIZE A)))

     (EQUAL (V (BV-ALU-CV A B C (NAT-TO-BV 9 4))) F)

     (EQUAL (BV (BV-ALU-CV A B C (NAT-TO-BV 1 4)))
            (INT-TO-BV
             (IF (INTEGER-IN-RANGEP (ADD (BV-TO-INT A) 1)
                                    (SIZE A))
                 (ADD (BV-TO-INT A) 1)
                 (IF (NEGATIVEP (ADD (BV-TO-INT A) 1))
                     (ADD (BV-TO-INT A)
                          (ADD 1 (EXP 2 (SIZE A))))
                     (ADD (BV-TO-INT A)
                          (ADD 1 (MINUS (EXP 2 (SIZE A)))))))
             (SIZE A)))

     (EQUAL (V (BV-ALU-CV A B C (NAT-TO-BV 1 4)))
            (NOT (INTEGER-IN-RANGEP (ADD (BV-TO-INT A) 1) (SIZE A))))
     ...)
```

The bit vector interpretation lemmas are similar in form to the interpretation lemmas presented above. This completes the formal instruction-level description of the FM8502.

Chapter 6
THE FM8502 DESIGN-LEVEL SPECIFICATION

The design-level specification for the FM8502 is an interpreter much like the instruction-level specification, but it concretely specifies a wiring diagram composed of gates, RAMs and a ROM. Most of the circuit descriptions describe wiring diagrams for circuits of an arbitrary size. This type of circuit description has enabled the use of induction in many circuit proofs, as we will see, and provides for very compact circuit descriptions.

In this section we first introduce our hardware model, then we examine a recursively specified circuit description. We then describe the FM8502 ALU which we follow with the actual machine description. The machine description actually models three cooperating processes: the FM8502 itself, the memory, and an oracle.

6.1 The Hardware Model

The FM8502 design-level is modeled with three types of elements: Boolean gates, random access memories (RAMs) and read-only memories (ROMs). We use functions to axiomatize each of these elements. Our design-level specifications are wiring diagrams which show how a set of our design-level elements are interconnected. The modeling of our concrete design-level elements is quite simple; we only model the logical properties of each element. This model is clearly too simple for manufacture. Issues such as delay, maximum fan-out, power consumption, implementation technology, etc., are not modeled.

We define a set of functions to axiomatize Boolean gates. Each gate takes one or two Boolean input(s) and returns a Boolean result.

```
(B-NOT  X)   = (IF X F T)
(B-AND  X Y) = (AND X Y)
(B-OR   X Y) = (OR X Y)
(B-NAND X Y) = (IF X (IF Y F T) T)
(B-NOR  X Y) = (IF X F (IF Y F T))
(B-XOR  X Y) = (IF X (IF Y F T) (IF Y T F))
(B-EQUV X Y) = (IF X (IF Y T F) (IF Y F T))
```

The functions **AND** and **OR** are defined in terms of **IF**. Each time a design-level specification has one of the Boolean connectives (gates), above, we think of this Boolean function as a hardware gate.

We model two types of RAMs with two functions and a state variable for each type. The state variable contains the contents of a RAM in question, and the two functions provide lookup and update operations. The two RAM types are a single register and an array of registers. **UPDATE-V** updates a single register; no single register look up function is required. The state component of a RAM is modeled as a list of bit vectors. (Earlier we defined **RAMP** which recognizes a well-formed RAM.) The function

V-NTH accesses a RAM and function **UPDATE-V-NTH** updates a RAM.

```
(NTH N LST) = (IF (ZEROP N)
                  (CAR LST)
                  (NTH (SUB1 N) (CDR LST)))

(V-NTH V-N LST) = (NTH (BV-TO-NAT V-N) LST)

(UPDATE-V C CELL VALUE) = (IF (TRUEP C) VALUE CELL)

(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)

(UPDATE-V-NTH C V-N LST VALUE)
 =
(UPDATE-NTH C (BV-TO-NAT V-N) LST VALUE)
```

We only use the update functions once for each state variable in the design-level specification. We think of the update functions representing a register or RAM with a write-enable input and a datum input. We think of a register as having an output that is just the contents of the register; we access this output in our design-level specifications by simply examining the state variable containing the contents of the register. The RAM has an additional (bit vector) input which is an address. This is apparent in the function **UPDATE-V-NTH**. The output of the RAM is accessed with **V-NTH**. We check at design-level expansion time (which we discuss later) to ensure the address used by the access function **V-NTH** is identical to the address given to the function **UPDATE-V-NTH**; otherwise, we would run the risk of having to implement a multi-port RAM.

ROMs are modeled by constant functions. There is only one ROM in the FM8502, and this ROM contains the micro-code program which the internal FM8502 machinery executes to interpret instruction-level instructions. We use the function **V-NTH** to access the ROM.

6.2 Hardware Functions

Many of the FM8502 design-level circuits are specified with recursive functions. Before presenting something the size of an ALU, we explain our recursive function specification method on a ripple-carry adder. Consider the function **BV-ADDER** below.

```
(BV-ADDER C A B)
 =
(IF (BTMP A)
    (BITV C (BTM))
    (BITV (B-XOR C (B-XOR (BIT A) (BIT B)))
          (BV-ADDER (B-OR (B-AND (BIT A) (BIT B))
                          (B-AND C (B-XOR (BIT A) (BIT B))))
                    (VEC A)
                    (VEC B))))
```

This function takes a one-bit carry input **C** and two bit vectors **A** and **B**. If the input bit vector **A** is empty, then the carry input is returned as a one-bit bit vector; otherwise, the three-way exclusive or is computed on the carry input and the first bit of **A** and **B** and **BV-ADDER** is called again with a new carry input and one-bit shorter bit vectors.

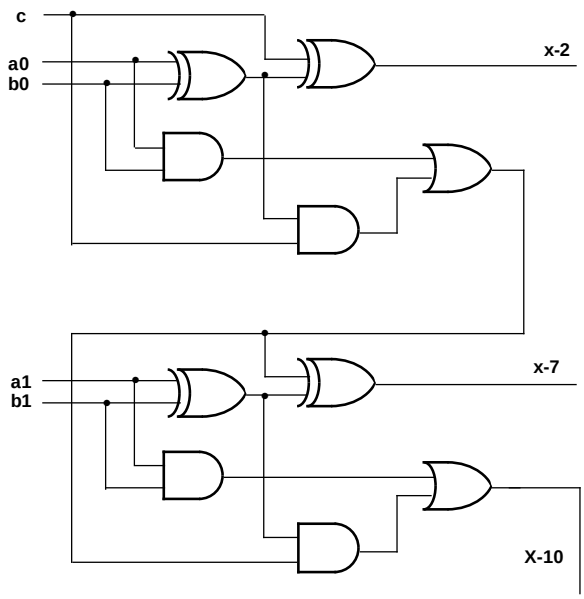

Figure 6-1: Two-bit Adder Schematic

A two-bit expansion of the function **BV-ADDER** produces 10 gates and is schematically pictured in Figure 6-1. We expand design-level circuit specifications by first identifying all common subexpressions in each function, because we only produce gates once for each common subexpression local to a particular function. We then repeatedly expand all functions, except our gate primitives, and collect the resulting gates into a theorem which is provably equivalent to the original design-level circuit specifications. This expander is a part of the Boyer-Moore theorem prover, which just expands functions using function definitions and rewrite rules from the Boyer-Moore data base. The resulting theorem from the two-bit expansion is shown below.

```
(IMPLIES
 (AND (AND (OR (FALSEP C) (TRUEP C))
           (OR (FALSEP A0) (TRUEP A0))
           (OR (FALSEP A1) (TRUEP A1))
           (OR (FALSEP B0) (TRUEP B0))
           (OR (FALSEP B1) (TRUEP B1)))
      (AND (EQUAL X-1 (B-XOR A0 B0))
           (EQUAL X-2 (B-XOR C X-1))
           (EQUAL X-3 (B-AND A0 B0))
           (EQUAL X-4 (B-AND C X-1))
           (EQUAL X-5 (B-OR X-3 X-4))
           (EQUAL X-6 (B-XOR A1 B1))
           (EQUAL X-7 (B-XOR X-5 X-6))
           (EQUAL X-8 (B-AND A1 B1))
           (EQUAL X-9 (B-AND X-5 X-6))
           (EQUAL X-10 (B-OR X-8 X-9))))
 (EQUAL (BV-ADDER C
                  (BITV A0 (BITV A1 (BTM)))
                  (BITV B0 (BITV B1 (BTM))))
        (BITV X-2 (BITV X-7 (BITV X-10 (BTM))))))
```

The correctness of **BV-ADDER** as it concerns natural number addition is presented below after the definition of **BV2P**. This lemma can be proven with induction, thus the **BV-ADDER** definition is a correct design-level specification for an *n*-bit ripple-carry adder.

```
(BV2P X Y) = (AND (BITVP X)
                  (BITVP Y)
                  (EQUAL (SIZE X) (SIZE Y)))


(IMPLIES (AND (BV2P A B)
              (BOOLP C))
         (EQUAL (BV-TO-NAT (BV-ADDER C A B))
                (PLUS (BV-TO-NAT A)
                      (BV-TO-NAT B)
                      (CARRY C))))
```

BV-ADDER returns a bit-vector one bit longer than the input bit vector. To make a fixed width bit vector adder, we strip off the last bit of the **BV-ADDER** result with **BV-ADDER-CARRY-OUT** and make it a carry output. We use the function **BV-ADDER-OUTPUT** to collect the first *n*-bits of **BV-ADDER**.

```
(BV-ADDER-OUTPUT C A B) = (TRUNC (BV-ADDER C A B) (SIZE A))

(BV-ADDER-CARRY-OUT C A B) = (BITN (BV-ADDER C A B) (ADD1 (SIZE A)))
```

Below are natural number interpretation lemmas for **BV-ADDER-OUTPUT** and **BV-ADDER-CARRY-OUT**. These interpretation lemmas are the same lemmas that are part of the natural number ALU interpretation lemma.

```
(IMPLIES
 (AND (BV2P A B)
      (BOOLP C))
 (EQUAL (BV-ADDER-OUTPUT C A B)
        (IF (LESSP (PLUS (BV-TO-NAT A) (BV-TO-NAT B) (CARRY C))
                   (EXP 2 (SIZE A)))
            (NAT-TO-BV (PLUS (BV-TO-NAT A) (BV-TO-NAT B) (CARRY C))
                       (SIZE A))
            (NAT-TO-BV
             (REMAINDER (PLUS (BV-TO-NAT A) (BV-TO-NAT B) (CARRY C))
                        (EXP 2 (SIZE A)))
             (SIZE A)))))

(IMPLIES
 (AND (BV2P A B)
      (BOOLP C))
 (EQUAL (BV-ADDER-CARRY-OUT C A B)
        (NOT (LESSP (PLUS (BV-TO-NAT A) (BV-TO-NAT B) (CARRY C))
                    (EXP 2 (SIZE A))))))
```

6.3 The ALU Specification

The FM8502 ALU design-level specification describes an *n*-bit ALU in much the same manner as **BV-ADDER**. This specification is quite large and we do not present it here; however, we sketch the idea of the ALU.

The FM8502 ALU design-level specification is composed of many design-level circuit specifications, such as **BV-ADDER**, and these are connected together with a number of selectors depending on the ALU op-code. The simple-minded approach taken with FM8501 was to have a 16-way selector being fed by 16 design-level circuit specifications. Although this works, it does not make best use of internal common subexpressions. The FM8502 ALU has selectors both near the output and input. The output selectors choose between various commonly used circuits. For example, an output selector might select between **BV-ADDER** and the bit vector or function **BV-OR**. The input selectors feed commonly used circuits with various inputs, e.g. **BV-ADDER** is used for adding, adding with carry, and

incrementing.

We prove the correctness of the FM8502 ALU by proving the various ALU interpretation lemmas correct. That is, we show that for bit vectors, natural numbers, and integers the ALU has "good" operational characteristics.

6.4 The Design-level Interpreter Specification

The FM8502 design-level specification describes the FM8502 on a micro-cycle by micro-cycle basis. This specification also includes an axiomatization for memory, which the FM8502 communicates with, and an oracle which supplies external events. The memory communicates with the FM8502 by shared connections and the oracle inputs are supplied as a list of values. Below is an abbreviated specification of the FM8502 design-level interpreter; this function "ticks" once per micro-instruction.

```
(FM8502-DESIGN m-state internal-state ORACLE)
 =
(LET ((NEXT-DESIGN-STATE
       (FM8502-DESIGN-STEP m-state internal-state (CAR ORACLE))))
  (IF (NLISTP ORACLE)
      (LIST m-state internal-state)
      (FM8502-DESIGN (CAR NEXT-DESIGN-STATE)
                     (CADR NEXT-DESIGN-STATE)
                     (CDR ORACLE))))

(FM8502-DESIGN-STEP m-state internal-state ORACLE-ELEMENT)
 =
(LIST (M-STATE (REG-FILE m-state internal-state)
               (C-FLAG   m-state internal-state)
               (V-FLAG   m-state internal-state)
               (N-FLAG   m-state internal-state)
               (Z-FLAG   m-state internal-state)
               (REAL-MEM m-state ORACLE-ELEMENT))
      (next-internal-state m-state internal-state
                           ORACLE-ELEMENT))
```

The functions above are more abstract than the actual FM8502 design-level specification. Functions and variables written in **CAPITALS** are actual design-level functions or state components while **lower case** is used to represent abstract design-level functions and state components.

If required, the function **REG-FILE** computes any update to the register file. Likewise, the functions **C-FLAG**, **V-FLAG**, **N-FLAG**, and **Z-FLAG** compute new values for the flag registers. The function **REAL-MEM** computes the contents of the memory. This function is actually separate from the FM8502, i.e., we do not expand this function when producing the FM8502 design-level wiring diagram. We consider the memory "process" external to the FM8502 microprocessor. The function **REAL-MEM** describes the operation of an acceptable memory.

The variable **ORACLE** provides external inputs to the FM8502. Each **ORACLE-ELEMENT** contains two values which model two external inputs: reset and data acknowledgment. Each time **FM8502-DESIGN** ticks, a new **ORACLE-ELEMENT** is given to **FM8502-DESIGN-STEP**. If the reset input is set, the FM8502 resets itself. If the data acknowledgment is set while the FM8502 is waiting for a response from memory, then the FM8502 proceeds. The proof of correctness for FM8502 obviously constrains the allowable **ORACLE**, e.g., the FM8502 will not execute any instructions if the reset input is always set.

Chapter 7
THE FM8502 GATE-GRAPH

The expansion of the FM8502 microprocessor into gates requires us to separate the memory process out of the FM8502 design-level specification. The FM8502 is conventional in that the processor itself includes a register file and flag registers but has an external memory. Extracting the memory process from the design-level specification and providing inputs for the oracle inputs reset and data acknowledgment leave the inputs and output pictured in Figure 7-1. The expansion of FM8502 produces several thousand Boolean gates; one 20-bit by 16 word ROM; 193 one-bit, write-enabled latches (registers); and one 32-bit by 8 word register file.

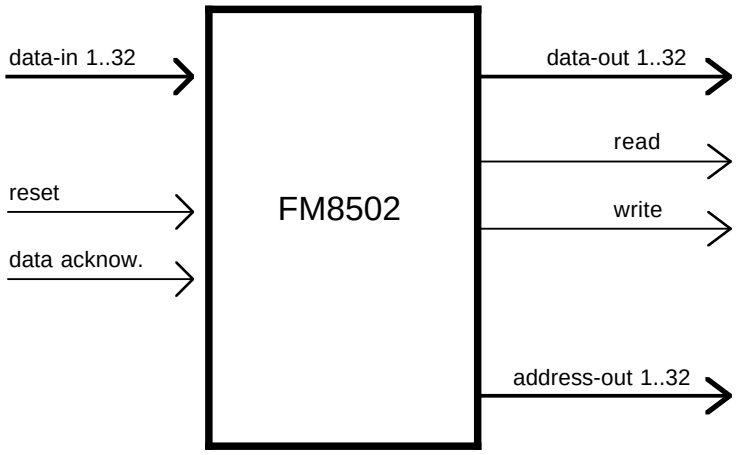

Figure 7-1: FM8502 Signal Groups

Chapter 8
THE FM8502 PROOF

The verification of the FM8502 is a multi-stage procedure. We begin by presenting the final FM8502 correctness theorems and then explain how we reached these final theorems. The correctness of the ALU lemmas are not described here as we have already discussed them.

8.1 The Final Theorem

The correctness of FM8502 was demonstrated by showing that the programmer-visible state computed by the FM8502 design-level specification is the same as the programmer-visible state computed by the FM8502 instruction-level specification. Since the design-level specification must execute many "clock cycles" before it completes one instruction-level instruction, we compute the necessary number of cycles the design-level interpreter needs for each instruction executed by the instruction-level interpreter. In the abstract statement of the proof we only collect the programmer-visible state from the design-level interpreter.

```
(IMPLIES (AND (M-STATEP PROG-STATE)
              (well-formed INTERNAL-STATE)
              (NUMBERP N))
         (EQUAL (FM8502 PROG-STATE N)
                (CAR (FM8502-DESIGN
                       PROG-STATE INTERNAL-STATE
                       (required-cycles PROG-STATE N
                                        MEMORY-DELAYS)))))
```

The theorem above says that the programmer-visible state computed by the instruction-level interpreter **FM8502** is identical to the programmer-visible state computed by the design-level interpreter **FM8502-DESIGN**, given that the design-level has the necessary number of clock ticks available.

The function **required-cycles** is a Skolem function which computes the number of clock cycles the design-level interpreter will need to execte to complete **N** instruction-level instructions, given the program and the various delays encountered by accessing memory. The variable **MEMORY-DELAYS** is just a list of delays for accessing memory; any values may be used for any element of this list. The number of cycles required by **FM8502-DESIGN** is equal to the length of the list generated by **require-cycles**; each element of this list contains an unset reset input and a data acknowledgement input. The times the data acknowledgement inputs are set is constructed from the values found in **MEMORY-DELAYS**.

We have also proved the correctness of the FM8502 during reset. This proof provides less information than the statement above. The FM8502 reset specification just says that the program counter will be zero after a reset. The proof that the FM8502 design does set the program counter to zero is

straightforward.

8.2 Organization of the Proof

The input required for the Boyer-Moore theorem prover to establish the correctness of the FM8502 is quite large. This input is a graduated sequence of more and more difficult lemmas intermixed with instruction-level and design-level specifications. The input is about a quarter of a million characters long, containing 216 definitions and 354 lemmas. The proof takes about 8 hours to process on a Sun 3/60HM-20 workstation running AKCL (Austin Kyoto Common Lisp). The number of definitions and proofs does not include theorem prover commands to enable and disable various definitions and lemmas, of which there are quite a number. This aspect of the proof is critical for success, as this is how the theorem prover's attention is focused on useful work.

The first part of the proof script contains various arithmetic definitions and lemmas. The majority of the lemmas involve facts about truncating arithmetic which are later used in establishing the correctness of the ALU interpretation lemmas. The functions we use to model Boolean gates are next introduced, followed by the axiomatization of bit vectors. Operations on bit vectors, such as **TRUNC** and **BITN**, come next, and these operations include bit vector exclusive or, bit vector not, etc.

The next major section of the proof script involves integers. Integer addition and subtraction are defined and a number of lemmas are proved about these functions.

Mapping functions between bit vectors, natural numbers, and integers specify the representation of naturals and integers with bit vectors. We prove some properties of the mapping functions and their compositions. At this time the correspondence between truncating natural number addition and truncating integer addition is established. This lemma, presented earlier, is proved without induction and splits into more than 450 cases. Most of these cases are dispatched by the prover using previously proved lemmas or its internal linear arithmetic package.

We next define **BV-ADDER**. We prove just two properties about **BV-ADDER** and disable its definition. These properties are: the length of the bit vector returned by **BV-ADDER** is one more than the length of its argument **A**; and **BV-ADDER** adds natural numbers when represented as bit vectors correctly. These two properties of **BV-ADDER** are the only two properties needed for the remainder of the proof script. Any function which satisfies these two properties can be used instead of **BV-ADDER**, which allows other adder circuits to be used.

We then define the fixed-width adder, the adder carry output, and the adder overflow output. We use these functions to define the subtractor also. With only the two properties above, we establish the natural number and integer operation for addition and subtraction. These lemmas later appear as part of the ALU interpretation lemmas.

We next define bit and bit vector selectors and prove they select properly. The definition of the ALU is next given and this is followed by the three ALU interpretation lemmas. Before attempting to prove the correctness of the ALU, we prove a lazy version of the ALU, which performs selection with the built-in function **IF**, equal to our actual design-level specification of the ALU. This involves using the properties about our bit and bit vector selectors. **IF** is not treated as a strict function when being considered by the theorem prover. Thus while proving interpretation properties for each different ALU op-code, only the necessary part of the ALU is examined. At this point we are 32% through the proof script.

We return to defining more of the design-level primitives by defining the memory access and update functions. We also define predicates which test RAMs, ROMs, and registers for proper structure.

The instruction set is specified by defining accessors which tear apart an instruction word into bit fields. The micro-ROM is defined along with a set of micro-ROM word field accessors. The "glue" logic required to control the various functional units in conjunction with the micro-code is defined next.

The memory process is defined along with a watch-dog process which axiomatizes the design-level specification of the memory. Using the glue logic functions and the memory specification, the FM8502 design-level specification is defined. We next prove several lemmas about the design-level specification which allow us to control this very large function in proofs. These lemmas describe the operation of the design-level interpreter with various oracles, and then the definition of the design-level interpreter is turned off.

We now begin a litany of lemmas about the micro-code, i.e., we prove lemmas describing the operation of the machine on a micro-cycle by micro-cycle basis. The statement of these lemmas is large because we are specifying what the entire machine does. The first part of the micro-code involves reseting the processor. The rest of the micro-code is arranged in a loop. Each time around the loop means one instruction-level instruction has been executed. The loop starts by fetching the instruction pointed to by the program counter. The processor waits for a data acknowledgment before reading its input port for the instruction from memory. The processor may have to wait an arbitrary amount of time. Thus just proving the processor can correctly fetch an instruction involves an induction with the entire state of the machine. The proof of the micro-code continues with fetching the operands, which requires the proof of a register operand fetch to be composed with an inductive memory access proof. This is effectively a branch in the micro-code, and the two paths rejoin later. However, different micro-code is not required because of some special internal logic which keeps the two cases straight. The micro-code proof proceeds by proving the sequencing of the operands through the ALU and onto their final destination. The possible post-increment operations are verified next and the micro-code loop is repeated.

The micro-code proofs are complicated by the need to ensure the total operation of the machine for every step. After proving the correctness of the micro-code step-by-step we concatenate the lemmas until we have a proof about one loop through the micro-code.

We now specify the instruction-level interpreter and again by using induction we are able to prove the final theorem presented above.

Chapter 9
CONCLUSIONS

The specification and verification of the FM8502 microprocessor has been performed with the aid of a heuristically guided mechanical theorem-prover. The formal specification of the FM8502 is far more compact than we find in commercial processor specifications and provides an unambiguous foundation for programs which use it. The formally specified design demonstrates the use of recursion for compactly specifying circuits with a high degree of regularity. The proof that the design-level specification correctly implements the instruction-level specification demonstrates the correctness of the FM8502 circuit design. The FM8502 design has been fully verified to implement its specification; this includes the specification and verification of every possible mix of instructions with every possible data value. The operation of FM8502 ALU has been verified to operate correctly no matter whether the operands are considered bit vectors, natural numbers, or integers.

Without the use of the Boyer-Moore theorem prover the proof of correctness would have been impossible due to its bulk. In addition, the tireless nature of a mechanical theorem prover prevents simple "human errors" from introducing themselves into proofs.[11]

The FM8502 has a much different *n*-bit design-level ALU specification than the one defined for the FM8501. This ALU, when expanded, requires only 1/5 as many gates as a 32-bit version of the FM8501 ALU. The new FM8502 ALU is proved to implement the same *n*-bit interpretation lemmas.

Except for the ALU, the 32-bit FM8502 was trivial to construct from the 16-bit FM8501 because both the design-level and instruction-level specifications describe an *n*-bit sized processor. The changes required to make an FM8502 only involved new instruction field operations along with straightforward changes to the instruction-level and design-level specifications. The structure of the FM8502 proof was similar to the structure of the FM8501 proof. To prove the correctness of the FM8502 for another word size only requires changing several constants and replaying the proof. It would be possible to prove the correctness of FM8502 for every word size wider than the 20-bit instruction, but the need to do this has not presented itself.

The FM8502 suffers from many of the same weaknesses of the FM8501. The characterization of external devices is still too intermixed with the design-level specifications. The difficulty of proving the micro-code correct remains unchanged; however, the expansion of circuit formulas into gate graphs has

---

[11]How does one know if the theorem prover is correct? One does not know for sure. But instead of having to check every proof presented we need only check the soundness of the theorem prover itself. Checking the theorem prover proceeds both with analysis and the social process of people using the theorem prover and attempting to find unsoundnesses. Literally, hundreds of people can use a mechanical theorem prover and check it for soundness. Very few people actually check proofs appearing in scientific presentations.

been improved greatly. Fan-out can now be explicitly specified on a function-by-function basis, whereas before there was no control of fan-out (or common subexpressions).

The use of a general-purpose logic for specifying the FM8502 allowed the top-level specification to be very abstract. For example, natural numbers are axiomatized with a Peano-like definition and addition is defined by concatenation. These definitions are simple, abstract, and believable. The ripple-carry adder function defines complex collections of Boolean gates which are not simple, abstract, nor believable, but these collections are correct.

This approach of specifying and mathematically proving the correctness of digital hardware seems to provide a method to control the complexity of ever-growing hardware circuits. Circuits are verified hierarchically, thus providing greater and greater levels of abstraction with each proof. The rigor imposed by a formal logic and proof ensures that with each layer no "hidden bugs" lay dormant for that yet untested case. This is especially important when stacking layers of definitions; bugs introduced at low levels can be extremely difficult to locate after they manifest themselves at high levels.

References

[BevierHuntYoung 87]
W.R. Bevier, W.A. Hunt, Jr., W.D. Young.
Toward Verified Execution Environments.
In *Proceedings of the 1987 Symposium on Security and Privacy*. IEEE, 1987.

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

[Cohn 89] Avra Cohn.
The Notion of Proof in Hardware Verification.
Journal of Automated Reasoning 5(2):127-139, June, 1989.

[Gordon 85] M. Gordon.
Why Higher-order Logic is a Good Formalism for Specifying and Verifying Hardware.
Technical Report 77, University of Cambridge, Computer Laboratory, 1985.

[HOL 87] M. Gordon.
HOL: A Proof Generating System for Higher-Order Logic.
Technical Report 103, University of Cambridge, Computer Laboratory, 1987.

[Hunt 85] W.A. Hunt, Jr.
FM8501: A Verified Microprocessor.
PhD Thesis, University of Texas at Austin, December, 1985.
Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703.

[Hunt 87] W. A. Hunt, Jr.
The Mechanical Verification of a Microprocessor Design.
In D. Borrione (editor), *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 89-132. North Holland, 1987.

[Joyce 88] Jeffrey J. Joyce.
Formal Specification and Verification of Asynchronous Processes in Higher-Order Logic.
Technical Report 136, University of Cambridge, Computer Laboratory, 1988.

[LCF_LSM 81] M. Gordon.
LCF_LSM.
Technical Report 41, University of Cambridge, Computer Laboratory, 1981.

Table of Contents

List of Figures

List of Tables