The Verification of a Bit-slice ALU

Warren A. Hunt, Jr.
Bishop C. Brock

Technical Report 49 September 1989

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

This paper was presented at the "Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects" at Cornell University July 1989, and will appear in "Lecture Notes in Computer Science".

Acknowledgements

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.

Abstract

The verification of a bit-slice ALU has been accomplished using a mechanical theorem prover. This ALU has an *n*-bit design specification, which has been verified to implement its top-level specification. The ALU and top-level specifications were written in the Boyer-Moore logic. The verification was carried out with the aid of Boyer-Moore theorem prover in a hierarchical fashion.

1. Introduction

The verification of a bit-slice ALU design has been accomplished with the aid of a mechanical theorem prover. This ALU, used in the FM8502 microprocessor [Hunt 89], has been proved to implement the FM8501 abstract ALU specification which precisely describes the operation of the FM8501 ALU in terms of natural numbers, integers, and bit vectors. This verification was accomplished in two steps: one, verifying that the FM8501 ALU implements its abstract specification, and two, verifying that the results computed by the FM8502 bit-slice ALU exactly match the results computed by the FM8501 ALU. This paper is concerned with only the second part of the verification; the first part has been documented elsewhere [Hunt 85].

Here we consider the abstract specification of the FM8502 ALU (henceforth referred to as the bit-slice ALU) to be the FM8501 ALU specification itself. The FM8501 ALU specification describes an n-bit ALU; that is, the specification of the FM8501 ALU describes how to construct an ALU of any size. The bit-slice ALU is also specified as an n-bit ALU, but requires far fewer gates.

The verification of the bit-slice ALU with respect to the FM8501 ALU represents an exhaustive Boolean comparison of the behavior of the two ALUs; i.e., the gate graphs specified by each ALU compute exactly the same results. The verification of Boolean functions with respect to other Boolean functions has recently received much attention. Randal Bryant [Bryant 86] has described a fast mechanism to compare Boolean circuits. This type of approach works well for many Boolean circuits, but is known to take exponentially increasing time with larger and larger multiplier circuits. Although Bryant's method is very fast, it does not allow specifications to be anything but Boolean functions.

Our approach involves using a general-purpose logic to represent our specifications. This allows the comparison of our Boolean circuits to abstract specifications containing, for example, integers, as well as comparing Boolean circuits to other Boolean circuits. Both ALU specifications are described as functions in the Boyer-Moore logic [Boyer & Moore 88]. The proof time to verify the equivalence of the two n-bit Boolean ALU specifications takes constant time. When a particular sized ALU is desired, then the ALU specifications functions are expanded (in linear time) into graphs of gates suitable for implementation.

The remainder of this paper begins with a summary of what we have proved, along with a quick introduction to our methodology. We then proceed in a bottom-up manner until we are able to present the specification of the two ALUs. The verification of the bit-slice ALU with respect to the FM8501 ALU is presented by describing how we organized our proof. This is followed by a comparison of the gate graphs generated from our two specifications.

2. Our Approach

The final theorem we prove is an equality correspondence theorem between the FM8501 and bit-slice ALUs. These two ALUs provide exactly the same functionality; our proof demonstrates their equality. In this section we sketch the ALU (equality) correspondence theorem before introducing our methodology. The proof methodology employed in establishing the ALU correspondence theorem is demonstrated by proving the correctness of two selector implementations.

2.1 A Look at the Final Theorem

The ALU correspondence theorem is presented below, written in the Lisp-style syntax of the Boyer-Moore logic. The FM8501 ALU is named **BV-ALU-CV** and the bit-slice ALU is named **NEW-ALU**. The correspondence theorem simply states that **BV-ALU-CV** and **NEW-ALU** are identical functions when **C** is a Boolean, **A** and **B** are bit vectors of the same length, and **OP-CODE** is a bit vector.[1]

```
(IMPLIES (AND (BOOLP C)
              (BV2P A B)
              (BITVP OP-CODE))

         (EQUAL (NEW-ALU C A B OP-CODE)
                (BV-ALU-CV A B C OP-CODE)))
```

This proof is one of Boolean equivalence; that is, the inputs are constrained to be Boolean or vectors of Booleans and the outputs are Booleans and a vector of Booleans.

An interesting facet of the ALU definitions is that they both describe *n*-bit ALUs. Our proof of their correspondence demonstrates the correctness of the **NEW-ALU** with respect to **BV-ALU-CV** for all word sizes. This type of verification requires induction, which is not found in Boolean decision procedures. To demonstrate our approach we present the verification of two selector implementations by induction.

2.2 The Boyer-Moore Logic and Theorem Prover

The Boyer-Moore logic [Boyer & Moore 88] is a quantifier-free, first-order predicate calculus with equality. Logic formulas are written in a prefix-style, Lisp-like notation. Included with the logic are several built-in data types: Booleans, natural numbers, lists, literal atoms, and integers.

The Boyer-Moore logic is unusual in that the logic may be extended by the application of any of the following axiomatic acts: defining conservative functions, adding recursively constructed data types, and adding arbitrary axioms. Adding an arbitrary formula as an axiom does not guarantee the soundness of the logic; we do not use this feature.

The Boyer-Moore theorem prover is a Common Lisp [Steele 84] program which provides a user with various commands to extend the logic and to prove theorems. The theorem prover is interactive and users enter commands through the top-level Common Lisp interpreter. The theorem prover manages a database of axioms, definitions, and proved theorems, thus allowing a user to concentrate on the less mundane aspects of proof development. The theorem prover contains decision procedures for tautology checking and linear arithmetic, a simplifier, and a rewriter. It is possible to add decision procedures to the theorem prover after proving their correctness.

We use the Boyer-Moore theorem prover as a proof checker. The theorem prover is led to difficult theorems by giving it a graduated sequence of more and more difficult lemmas until a final result can be obtained.

[1] Later we present definitions for **BOOLP**, **BV2P**, **BITVP**, **NEW-ALU**, and **BV-ALU-CV**.

2.3 Bit Vectors

Bit vectors are axiomatized by adding a new Boyer-Moore data type. Bit vectors are defined recursively, and each bit vector constructor function takes a bit and a bit vector as arguments. We employ the Boyer-Moore Shell Principle to formally define the bit vector data type.

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`**.

Some of the axioms introduced as a result of this data type definition are below.

```
(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 Y)
         (EQUAL (VEC (BITV X Y)) Y))
```

We define our bit vectors to have a "little-endian" format; thus the number 6 can be represented by **`(BITV F (BITV T (BITV T (BTM))))`**. We define the function **`NAT-TO-BV`** to convert a natural number **`N`** into a bit vector of **`SIZE`** bits.

```
(NAT-TO-BV N SIZE)
 =
(IF (ZEROP SIZE)
    (BTM)
    (BITV (NOT (ZEROP (REMAINDER N 2)))
          (NAT-TO-BV (QUOTIENT N 2) (SUB1 SIZE)))))
```

We define several functions which are useful when working with bit vectors. The function **`BTMP`**, defined below, formalizes our notion of an empty bit vector. If **`X`** is recognized as a bit vector, then **`X`** is empty if **`X = (BTM)`**; otherwise, **`X`** is considered empty.

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

(SIZE X) = (IF (BTMP X)
               0
               (ADD1 (SIZE (VEC X)))))
```

`SIZE` computes the size of a bit vector as follows: an empty bit vector has size **`0`**; otherwise, the bit vector has a size one greater than the size of the **`VEC`** of the bit vector. **`SIZE`** is a recursive function; we will see many functions which recur on the structure of a bit vector.

Bit vectors are appended with the function **`V-APPEND`**, which operates in a manner analogous to a list append function. The function **`BITN`** selects a particular bit from bit vector **`X`** given an index **`N`**. **`TRUNC`** truncates (or extends with **`F`**'s) bit vector **`A`** to a size **`N`**.

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

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

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

The function **BOOLP** is defined to recognize Boolean valued objects. We have defined the predicate **BV2P** which recognizes two bit vectors of identical size. This predicate is often used in the hypothesis of theorems which state properties about functions which operate upon two bit vectors simultaneously.

```
(BOOLP X) = (OR (FALSEP X) (TRUEP X)))

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

2.4 Hardware Primitives

We use functions to formalize our notion of combinational logic. We do not formalize registers or memory devices here, as our two ALUs are purely combinational.

```
(B-BUF X)             =  (IF X T F)
(B-NOT X)             =  (NOT X)

(B-NAND  A B)         =  (NOT (AND A B))
(B-NAND3 A B C)       =  (NOT (AND A B C))
(B-NAND4 A B C D)     =  (NOT (AND A B C D))

(B-OR   A B)          =  (OR A B)
(B-OR3  A B C)        =  (OR A B C)
(B-OR4  A B C D)      =  (OR A B C D)

(B-EQUV X Y)          =  (IF X (IF Y T F) (IF Y F T))
(B-XOR  X Y)          =  (IF X (IF Y F T) (IF Y T F))

(B-AND  A B)          =  (AND A B)
(B-AND3 A B C)        =  (AND A B C)
(B-AND4 A B C D)      =  (AND A B C D)

(B-NOR  A B)          =  (NOT (OR A B))
(B-NOR3 A B C)        =  (NOT (OR A B C))
(B-NOR4 A B C D)      =  (NOT (OR A B C D))
```

We think of the functions above as representing primitive gates, e.g., gate array macro cells. When we expand circuit implementation specifications, we do not expand the definitions of the above functions.

For convenience we have defined several other gate functions; these functions are not primitives but are defined in terms of gate primitives. These functions are listed below.

```
(B-XOR3 A B C)        = (B-XOR (B-XOR A B) C)
(B-XOR4 A B C D)      = (B-XOR (B-XOR A B) (B-XOR C D))

(B-AND5 A B C D E)    = (B-AND (B-AND3 A B C) (B-AND D E))
(B-AND6 A B C D E G)  = (B-AND (B-AND3 A B C) (B-AND3 D E G))
```

2.5 A Hardware Verification Methodology

The hardware verification methodology we employ involves using the Boyer-Moore logic to record abstract specifications and design specifications. Abstract specifications represent what we think of as "obviously" correct specifications. Design specifications describe implementations in terms of graphs of gates. Often we use recursive functions to describe *n*-bit implementations. We first present our methodology in general; we later narrow our focus to the comparison of Boolean functions.

To demonstrate our methodology, we consider the specification and verification of a selector. Our hardware selector selects one of two inputs for output. The value of the output is controlled by a third input. Abstractly, we specify a selector with inputs **A** and **B** and control input **C** by the following expression.

```
(IF C A B)
```

This specification is more abstract than combinational hardware implementations permit; for instance, **A** could be **7**.

Before giving the definition of our bit-vector selector, we investigate the Boolean selector defined below.

```
(B-IF C A B)
 =
(B-NAND (B-NAND C A)
        (B-NAND (B-NOT C) B))
```

This formal design specification can also be represented schematically as shown in Figure 1. The proof of

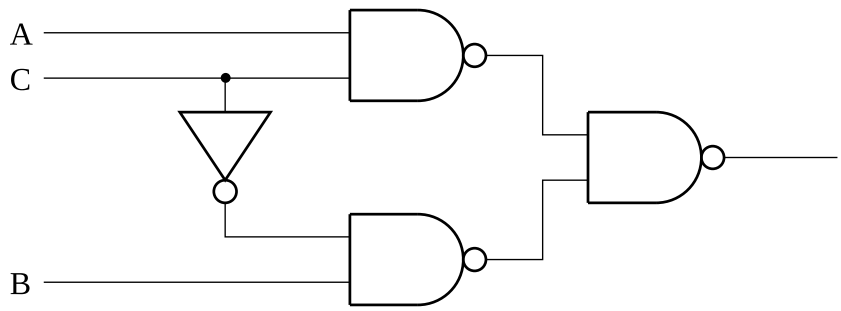

Figure 1: One-bit Selector Circuit

correctness of the one-bit selector we state as follows.

```
(IMPLIES (AND (BOOLP A)
              (BOOLP B)
              (BOOLP C))
         (EQUAL (B-IF C A B)
                (IF C A B)))
```

This theorem says the one-bit selector **B-IF** implements our abstract specification when **A**, **B**, and **C** are Boolean.

Our bit-vector selector implementation assumes that **A** and **B** are bit vectors of the same size, and that **C** is a Boolean input. Our bit vector selector is composed of some number of bit selectors. Using a recursive function we specify an *n*-bit selector named **BV-IF**.

```
(BV-IF C A B) = (IF (BTMP A)
                    (BTM)
                    (BITV (B-IF C (BIT A) (BIT B))
                          (BV-IF C (VEC A) (VEC B))))
```

The function **BV-IF** recurs when **A** is not empty. The proof that **BV-IF** implements our abstract specification is stated below.

```
(IMPLIES (AND (BOOLP C)
              (BV2P A B))
         (EQUAL (BV-IF C A B)
                (IF C A B)))
```

This theorem is proved by induction. The proof of this theorem can be carried out by the theorem prover automatically. We present the theorem prover input and its output for this proof. The theorem prover command for a proof attempt is **PROVE-LEMMA**, the name of this event is **BV-IF-WORKS**, and the lemma type is **REWRITE**. We have also given two heuristic hints (**INDUCT** and **DISABLE**) which are technically unnecessary but make the output more compact.

```
(PROVE-LEMMA BV-IF-WORKS (REWRITE)
  (IMPLIES (BV2P A B)
           (EQUAL (BV-IF C A B)
                  (IF C A B)))
  ((INDUCT (BV-IF C A B))
   (DISABLE SIZE)))

This formula can be simplified, using the abbreviations BTMP,
BV2P, IMPLIES, NOT, OR, and AND, to the following two new
formulas:

Case 2. (IMPLIES (AND (BTMP A)
                      (BITVP A)
                      (BITVP B)
                      (EQUAL (SIZE A) (SIZE B)))
                 (EQUAL (BV-IF C A B) (IF C A B))).

  This simplifies, rewriting with SIZE-BOTTOM, and unfolding the
  definitions of BTMP, BITVP, SIZE, BV-IF, and EQUAL, to:

        T.

Case 1. (IMPLIES (AND (BITVP A)
                      (NOT (EQUAL A (BTM)))
                      (IMPLIES (BV2P (VEC A) (VEC B))
                               (EQUAL (BV-IF C (VEC A) (VEC B))
                                      (IF C (VEC A) (VEC B))))
                      (BITVP B)
                      (EQUAL (SIZE A) (SIZE B)))
                 (EQUAL (BV-IF C A B) (IF C A B))),

  which simplifies, appealing to the lemmas BV2P-VEC,
  BITV-BIT-VEC, and B-IF-WORKS, and expanding the definitions
  of BV2P, IMPLIES,   TRUEP, BOOLP, BTMP, and BV-IF, to the goal:
```

```
        (IMPLIES (AND (BITVP A)
                      (NOT (EQUAL A (BTM)))
                      (NOT C)
                      (EQUAL (BV-IF C (VEC A) (VEC B))
                             (VEC B))
                      (BITVP B)
                      (EQUAL (SIZE A) (SIZE B)))
                 (EQUAL (BV-IF F A B) B)).

  However, this again simplifies, applying the lemmas BITV-BIT-VEC
  and B-IF-WORKS, and expanding the definitions of TRUEP, BOOLP,
  BTMP, and BV-IF, to:

        (IMPLIES (AND (BITVP A)
                      (NOT (EQUAL A (BTM)))
                      (EQUAL (BV-IF F (VEC A) (VEC B))
                             (VEC B))
                      (BITVP B)
                      (EQUAL (SIZE A) (SIZE B))
                      (EQUAL B (BTM)))
                 (EQUAL (BITV F (BTM)) B)).

  This again simplifies, rewriting with the lemma SIZE-BOTTOM, and
  opening up the functions VEC, BITVP, SIZE, and BTMP, to:

        T.

Q.E.D.

[ 0.1 1.1 0.4 ]
BV-IF-WORKS
```

To produce a gate graph from a recursive circuit specification, for instance **BV-IF**, we symbolically expand the circuit specification on symbolic inputs. Before expanding a function we identify any common subexpressions present; each common expression is expanded only once and its output is fanned out.

The function **BV-IF** contains only one common subexpression in its body, namely **C**. We can observe that the input **C** will be fanned out to every call of **B-IF**, because **C** is recursively passed on without being buffered. Let us proceed through the expansion of **BV-IF** into a two-bit selector. Our expansion process assumes that all free variables are constrained to be Boolean; we make this explicit later.

To begin our symbolic expansion of **BV-IF** we instantiate the formal arguments of a call to **BV-IF** with a control input and two, two-bit bit vectors. By replacing the call of **BV-IF** with its definition we obtain the second expression below. The third expression below is a simplification of the second after observing that **(BTMP (BITV A0 (BITV A1 (BTM))))** is false and using facts about **BIT**, **VEC**, and **BITV**.

```
(BV-IF C (BITV A0 (BITV A1 (BTM))) (BITV B1 (BITV B1 (BTM))))
 =
(IF (BTMP (BITV A0 (BITV A1 (BTM))))
    (BTM)
    (BITV (B-IF C (BIT (BITV A0 (BITV A1 (BTM))))
                  (BIT (BITV B1 (BITV B1 (BTM)))))
          (BV-IF C (VEC (BITV A0 (BITV A1 (BTM))))
                   (VEC (BITV B1 (BITV B1 (BTM)))))))
 =
(BITV (B-IF C A0 B0)
      (BV-IF C (BITV A1 (BTM)) (BITV B1 (BTM))))
```

We now expand **BV-IF** again, simplify, and get the next expression. The remaining call of **BV-IF** is simplified to just **BTM**.

```
(BITV (B-IF C A0 B0)
      (BITV (B-IF C A1 B1)
            (BV-IF C (BTM) (BTM))))
 =
(BITV (B-IF C A0 B0)
      (BITV (B-IF C A1 B1)
            (BTM)))
```

We now expand the definition of **B-IF** to obtain the following expression.

```
(BITV (B-NAND (B-NAND C A0)
              (B-NAND (B-NOT C) B0))
      (BITV (B-NAND (B-NAND C A1)
                    (B-NAND (B-NOT C) B1))
            (BTM)))
```

The expansion process presented above is somewhat simplified. We did not need to explicitly identify any common subexpression other than **C**, thus we did not need to share any active circuitry (gates). Our mechanical expansion process actually produces a theorem where the input variables (**C**, **A0**, **A1**, **B0**, **B1**) are constrained to be Boolean. For example, the theorem produced for the two-bit selector is given 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-NAND C A0))
                   (EQUAL X-2 (B-NOT C))
                   (EQUAL X-3 (B-NAND X-2 B0))
                   (EQUAL X-4 (B-NAND X-1 X-3))
                   (EQUAL X-5 (B-NAND C A1))
                   (EQUAL X-6 (B-NOT C))
                   (EQUAL X-7 (B-NAND X-6 B1))
                   (EQUAL X-8 (B-NAND X-5 X-7))))
         (EQUAL (BV-IF C
                       (BITV A0 (BITV A1 (BTM)))
                       (BITV B0 (BITV B1 (BTM))))
                (BITV X-4 (BITV X-8 (BTM)))))
```

Common subexpressions are named and placed in the hypothesis of the theorem. Note, however, that the term **(B-NOT C)** occurs twice. The expander only collects the common subexpressions which appear in the body of a function definition and not those which manifest themselves during the expansion process. It is possible to submit this theorem back to theorem prover and have it prove it.

2.6 A Narrow View of Circuit Verification

The abstract specification for our bit-slice ALU is the FM8501 ALU. As mentioned previously, more abstract specifications for the ALUs can be found elsewhere [Hunt 85, Hunt 87]. Here we are interested in demonstrating the precise correspondence of two Boolean functions without concerning ourselves with their more general properties. Both ALU functions produce a Boolean carry output, a Boolean overflow output, and an *n*-bit bit vector result, and we want to prove the exact correspondence of these two design specifications. Part of the verification is similar to other methods of Boolean function verification; however, proving the correctness of *n*-bit circuit specifications requires the use of induction.

To demonstrate our ALU verification approach, we define a new selector implementation and verify it with respect to **BV-IF**. As an aside, once we have verified that our new selector specification is identical to **BV-IF**, then we know this new selector satisfies our original abstract selector specification.

We begin by presenting the definition of our new selector function **BV4-IF**. This function is more complicated than **BV-IF** for several reasons: the circuit fan-out is limited, it recurs four bits at a time, and it contains a larger case structure. The helper function **B-IF-BAR** is used in **BV4-IF**.

```
(B-IF-BAR C C-BAR A B)
 =
(B-NAND (B-NAND C A)
        (B-NAND C-BAR B))


(BV4-IF C A B)
 =
(LET ((A0 (BIT A))
      (A1 (BIT (VEC A)))
      (A2 (BIT (VEC (VEC A))))
      (A3 (BIT (VEC (VEC (VEC A)))))
      (B0 (BIT B))
      (B1 (BIT (VEC B)))
      (B2 (BIT (VEC (VEC B))))
      (B3 (BIT (VEC (VEC (VEC B)))))
      (C     (B-BUF C))
      (C-BAR (B-NOT C)))

     (COND
      ((BTMP A) (BTM))

      ((BTMP (VEC A))
       (BITV (B-IF-BAR C C-BAR A0 B0) (BTM)))

      ((BTMP (VEC (VEC A)))
       (BITV (B-IF-BAR C C-BAR A0 B0)
             (BITV (B-IF-BAR C C-BAR A1 B1)
                   (BTM))))

      ((BTMP (VEC (VEC (VEC A))))
       (BITV (B-IF-BAR C C-BAR A0 B0)
             (BITV (B-IF-BAR C C-BAR A1 B1)
                   (BITV (B-IF-BAR C C-BAR A2 B2)
                         (BTM)))))
      (T (BITV
          (B-IF-BAR C C-BAR A0 B0)
          (BITV
           (B-IF-BAR C C-BAR A1 B1)
           (BITV
            (B-IF-BAR C C-BAR A2 B2)
            (BITV
             (B-IF-BAR C C-BAR A3 B3)
             (BV4-IF C
                     (VEC (VEC (VEC (VEC A))))
                     (VEC (VEC (VEC (VEC B))))))))))))
```

The first part of the **BV4-IF** definition contains abbreviations for the first four bits of the two input vectors and buffered control inputs; these abbreviations are used in the main body of the definition. This definition considers five cases depending on the size of **A**. For the cases where the size of **A** is three bits or less, this definition describes fixed-sized gate graphs. When the size of **A** is greater than four, this function recurs.

To prove the equivalence of these two functions, we prove the following lemma.

```
(IMPLIES (BV2P A B)
         (EQUAL (BV4-IF C A B)
                (BV-IF C A B)))
```

We prove this lemma by first proving the following two lemmas with induction. The first lemma says, if bit vector **A** has size **N**, then **(TRUNC A N)** is just **A**. The second lemma describes the operation of **BV4-IF** on any two bit vectors.

```
(IMPLIES (AND (EQUAL (SIZE A) N)
              (BITVP A))
         (EQUAL (TRUNC A N) A))

(IMPLIES (AND (BITVP A)
              (BITVP B))
         (EQUAL (BV4-IF C A B)
                (IF C A (TRUNC B (SIZE A)))))
```

By chaining these two lemmas together along with our lemma above describing the operation of **BV-IF**, we can prove that the two selectors work identically.

It is interesting to compare the implementations specified by the two different selector functions. The items we compare are gate count, fanout, and delay: gate count is the number of primitive gates required for the circuit; fanout is specified as the maximum number of gate inputs driven by any gate primitive or any input; and delay is the maximum number of gate delays encountered by all input signals where each gate primitive has unit delay. Below is a table comparing three differently sized expansions of our two selectors.

Selector Gate Graph Comparison						
Selector Size in Bits	BV-IF Gate Count	BV-IF Fan-out	BV-IF Delay	BV4-IF Gate Count	BV4-IF Fan-out	BV4-IF Delay
8-bits	32	16	3	28	6	4
16-bits	64	32	3	56	6	6
32-bits	128	64	3	112	6	10

We see **BV-IF** has a delay of three, but has a very large fanout. **BV4-IF** fanout is limited to six, but the delay is longer because of the buffering of the control line.

This concludes the introduction to our method. The remainder of this paper is a presentation of the FM8501 and bit-slice ALUs, the verification of the bit-slice ALU with respect to the FM8501 ALU, and a comparison of the two implementations.

3. The FM8501 ALU Specification

The FM8501 ALU is composed of a number of functional units connected by a large selector. The FM8501 ALU is a 16-function ALU with logical, shift, addition, and subtraction operations which are summarized in Table 1. We first present the various functional units and then the ALU definition itself.

Several of the ALU operations are simple logical operations. For each of these logical operations we have defined a hardware implementation specification. Each of these specifications operate on *n*-bit inputs. **BV-NOT** provides logical negation. **BV-AND**, **BV-OR**, and **BV-XOR** are the logical and, logical or, and exclusive or functions, respectively.[2]

---

[2]Note: The definitions presented below for the FM8501 ALU specification are syntactically slightly different than previously presented [Hunt 85]; however, they are semantically the same.

OP-CODE	Result	Description
0000	a	Move
0001	$a+1$	Increment
0010	$b+a+c$	Add with carry
0011	$b+a$	Add
0100	$0-a$	Negation
0101	$a-1$	Decrement
0110	$b-a-c$	Subtract with borrow
0111	$b-a$	Subtract
1000	$a>>1$	Rotate right, shifted through carry
1001	$a>>1$	Arithmetic shift right, top bit duplicated
1010	$a>>1$	Logical shift right, top bit zero
1011	$b \veebar a$	Exclusive or
1100	$b \vee a$	Or
1101	$b \wedge a$	And
1110	$\neg a$	Not
1111	a	Move

Table 1: FM8501 ALU Operation Summary

```
(BV-NOT X) = (IF (BTMP X)
                 (BTM)
                 (BITV (B-NOT (BIT X))
                       (BV-NOT (VEC X))))

(BV-AND X Y) = (IF (BTMP X)
                   (BTM)
                   (BITV (B-AND (BIT X) (BIT Y))
                         (BV-AND (VEC X) (VEC Y))))

(BV-OR X Y) = (IF (BTMP X)
                  (BTM)
                  (BITV (B-OR (BIT X) (BIT Y))
                        (BV-OR (VEC X) (VEC Y))))

(BV-XOR X Y) = (IF (BTMP X)
                   (BTM)
                   (BITV (B-XOR (BIT X) (BIT Y))
                         (BV-XOR (VEC X) (VEC Y))))
```

There are three right shift operations: logical shift right, **BV-LSR**; arithmetic shift right, **BV-ASR**; and rotate right with carry, **BV-ROR**. Left shift operations can be provided by the adder.

```
(BV-LSR A) = (IF (BTMP A)
                 (BTM)
                 (V-APPEND (VEC A)
                           (BITV F (BTM))))

(BV-ASR A) = (IF (BTMP A)
                 (BTM)
                 (V-APPEND (VEC A)
                           (BITV (BITN A (SIZE A)) (BTM))))

(BV-ROR A C) = (IF (BTMP A)
                   (BTM)
                   (V-APPEND (VEC A)
                             (BITV C (BTM))))
```

The addition and subtraction functions are all composed from the function **BV-ADDER**. Typically, hardware adders return a bit vector of the same size as their inputs; however, **BV-ADDER** is a specification of a ripple-carry adder which returns a bit vector whose size is one bit longer than its **A** formal parameter. We define three functions for addition and subtraction which, given two *n*-bit inputs and a carry input, provide an *n*-bit output, a Boolean overflow output, and a Boolean carry-out output. **BV-ADDER-OUTPUT** truncates the output of **BV-ADDER** to the size of the **A** bit vector. **BV-ADDER-CARRY-OUT** returns the carry-out by selecting the most significant bit of **BV-ADDER**. **BV-ADDER-OVERFLOWP** computes a Boolean result which is true when integer addition does not provide the correct answer.

```
(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-OR (B-AND (BIT A) C)
                                (B-AND (BIT B) C)))
                    (VEC A)
                    (VEC B))))


(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)))

(BV-ADDER-OVERFLOWP C A B)
 =
(B-AND (B-EQUV (BITN A (SIZE A)) (BITN B (SIZE B)))
       (B-XOR (BITN A (SIZE A))
              (BITN (BV-ADDER-OUTPUT C A B)
                    (SIZE A))))


(BV-SUBTRACTER-OUTPUT C A B)
 =
(BV-ADDER-OUTPUT (B-NOT C) (BV-NOT A) B)

(BV-SUBTRACTER-CARRY-OUT C A B)
 =
(B-NOT (BV-ADDER-CARRY-OUT (B-NOT C) (BV-NOT A) B))

(BV-SUBTRACTER-OVERFLOWP C A B)
 =
(BV-ADDER-OVERFLOWP (B-NOT C) (BV-NOT A) B)
```

Similarly, **BV-SUBTRACTER-OUTPUT** defines a subtracter function with an *n*-bit output given an *n*-bit input. **BV-SUBTRACTER-CARRY-OUT** outputs **T** when a borrow is required, else **F**. **BV-SUBTRACTER-OVERFLOWP** computes whether an integer subtraction overflowed.

The FM8501 ALU returns three results: a bit vector result, a carry-out, and an overflow. We define a new data type, with constructor **BV-CV**, which simply packages a bit vector and two Booleans into one object. Below is a hardware selector function for objects of this new data type. The **BV-CV-IF** function selects between two **BV-CV** objects, **A** and **B**, by accessing their components, with accessors **BV**, **C**, and **V**; using **B-IF** and **BV-IF**; and recombining the selector results with **BV-CV**.

```
(BV-CV-IF C A B)
 =
(BV-CV (BV-IF C (BV A) (BV B))
       (B-IF  C (C  A) (C  B))
       (B-IF  C (V  A) (V  B)))

(IMPLIES
 (BV2P AV BV)
 (EQUAL (BV-CV-IF C (BV-CV AV AC AO) (BV-CV BV BC BO))
        (IF       C (BV-CV AV AC AO) (BV-CV BV BC BO))))
```

The lemma just above gives an abstract view of the **BV-CV-IF** selector function.

With the definitions above, it is now possible to define the FM8501 ALU. It takes a carry input, **C**, two bit vector inputs, **A** and **B**, and a 4-bit operation code, **OP-CODE**. This ALU computes all three outputs, bit vector, carry, and overflow, for every operation code.

```
(BV-ALU-CV A B C OP-CODE)
 =
(LET ((BV-ZERO (NAT-TO-BV 0 (SIZE A))))
  (BV-CV-IF (BITN OP-CODE 4)
    (BV-CV-IF (BITN OP-CODE 3)
      (BV-CV-IF (BITN OP-CODE 2)
        (BV-CV-IF (BITN OP-CODE 1)
         (BV-CV A F F)                          ; op-code 15 - move
         (BV-CV (BV-NOT A) F F))                ; op-code 14 - not
        (BV-CV-IF (BITN OP-CODE 1)
         (BV-CV (BV-AND A B) F F)               ; op-code 13 - and
         (BV-CV (BV-OR A B) F F)))              ; op-code 12 - or
      (BV-CV-IF (BITN OP-CODE 2)
        (BV-CV-IF (BITN OP-CODE 1)
         (BV-CV (BV-XOR A B) F F)               ; op-code 11 - xor
         (BV-CV (BV-LSR A) (BITN A 1) F))       ; op-code 10 - lsr
        (BV-CV-IF (BITN OP-CODE 1)
         (BV-CV (BV-ASR A) (BITN A 1) F)        ; op-code 9 - asr
         (BV-CV (BV-ROR A C)                    ; op-code 8 - ror
                (IF (ZEROP (SIZE A))
                    C
                  (BITN A 1))
                F))))
    (BV-CV-IF (BITN OP-CODE 3)
      (BV-CV-IF (BITN OP-CODE 2)
        (BV-CV-IF (BITN OP-CODE 1)
         (BV-CV (BV-SUBTRACTER-OUTPUT F A B)     ; op-code 7 - sub
                (BV-SUBTRACTER-CARRY-OUT F A B)
                (BV-SUBTRACTER-OVERFLOWP F A B))
         (BV-CV (BV-SUBTRACTER-OUTPUT C A B)     ; op-code 6 - subb
                (BV-SUBTRACTER-CARRY-OUT C A B)
                (BV-SUBTRACTER-OVERFLOWP C A B)))
        (BV-CV-IF (BITN OP-CODE 1)               ; op-code 5 - dec
         (BV-CV (BV-SUBTRACTER-OUTPUT T BV-ZERO A)
                (BV-SUBTRACTER-CARRY-OUT T BV-ZERO A)
                (BV-SUBTRACTER-OVERFLOWP T BV-ZERO A))
                                                 ; op-code 4 - neg
         (BV-CV (BV-SUBTRACTER-OUTPUT F A BV-ZERO)
                (BV-SUBTRACTER-CARRY-OUT F A BV-ZERO)
                (BV-SUBTRACTER-OVERFLOWP F A BV-ZERO))))
```

```
(BV-CV-IF (BITN OP-CODE 2)
  (BV-CV-IF (BITN OP-CODE 1)
   (BV-CV (BV-ADDER-OUTPUT F A B)         ; op-code 3 - add
          (BV-ADDER-CARRY-OUT F A B)
          (BV-ADDER-OVERFLOWP F A B))
   (BV-CV (BV-ADDER-OUTPUT C A B)         ; op-code 2 - addc
          (BV-ADDER-CARRY-OUT C A B)
          (BV-ADDER-OVERFLOWP C A B)))
 (BV-CV-IF (BITN OP-CODE 1)               ; op-code 1 - inc
  (BV-CV (BV-ADDER-OUTPUT T A BV-ZERO)
         (BV-ADDER-CARRY-OUT T A BV-ZERO)
         (BV-ADDER-OVERFLOWP T A BV-ZERO))
  (BV-CV A F F))))))
```

It is this definition of the FM8501 ALU that we prove equivalent to the bit-slice ALU.

4. The Bit-slice ALU Specification

The bit-slice ALU definition is similar in structure to a conventionally designed ALU. This ALU has a bit-slice implementation and uses a propagate and generate structure for computing carries in each slice. Carry propagation between each 3-bit slice is serial.

Schematically, our new ALU is constructed as shown in Figure 2. The **`Shift or Buffer`** module either buffers the input or performs one of three right shifts, as prescribed by **`Decode shift`**. The **`3-Bit-by-3-bit recursive ALU`** is the bit slice ALU which provides the functions besides right shifts. **`Carry Help`** and **`Overflow Help`** adjust the output of the ALU to conform with the FM8501 ALU.

```
op-code =====+==========+============+===================+
             |          |            |                   |
             V          |            V                   |
       --------------   |    -------------------         V
       |Decode shift|   |    |      Decode     |     ---------
       |------------|   |    | Prop, Gen, Mode |     | Carry |
  a ===>|            |   |    |-----------------|  c ->| Out   |-->
       | Shift or   |        |                 |     | Help  |
       |  Buffer    |======>|                 |     |-------|
  c -+->|            |        |                 |     |       |
     |  --------------   |    |  3-Bit-by-3-bit |=====>|       |==>
     |                   V    |  recursive      |     |       |
     |   -----------------    |  ALU            |     |-------|
     +-->| Carry In Help |-->|                 | an ->| Over- |
         -----------------    |                 | bn ->| flow  |-->
                              |                 |     | Help  |
  b =======================>|                 |     ---------
                              -------------------
```

Figure 2: Bit-slice ALU Structure

We define the bit-slice ALU in pieces, starting with the shift unit. This shift unit provides three kinds of right shifts or buffers its input. We define **`BV-SHIFT-RIGHT`** to right shift **`A`** one bit, with the most significant bit now being **`SI`**.

```
(BV-SHIFT-RIGHT A SI) = (IF (BTMP A)
                            (BTM)
                            (BV-APP (VEC A)
                                    (BITV SI (BTM)))))

(BV-SHIFT-OR-BUF C A OP-CODE)
 =
(LET ((OP1 (B-BUF (BITN OP-CODE 1)))
      (OP2 (B-BUF (BITN OP-CODE 2)))
      (OP3 (B-BUF (BITN OP-CODE 3)))
      (OP4 (B-BUF (BITN OP-CODE 4))))

  (LET ((DECODE-BUF (B-OR (B-AND OP1 OP2)
                          (B-NAND (B-NOT OP3) OP4)))
        (DECODE-ASR (B-AND4 OP1 (B-NOT OP2) (B-NOT OP3) OP4))
        (DECODE-ROR (B-AND4 (B-NOT OP1) (B-NOT OP2)
                            (B-NOT OP3) OP4)))

    (BV4-IF DECODE-BUF A
            (BV-SHIFT-RIGHT A (B-OR (B-AND DECODE-ASR
                                           (BITN A (SIZE A)))
                                    (B-AND DECODE-ROR C))))))
```

The function **BV-SHIFT-OR-BUF** provides buffering for the input operation code and decodes when this function should buffer, arithmetic shift right, or rotate right. The decoding logic controls the functions **BV4-IF** and **BV-SHIFT-RIGHT** to provide the desired functionality.

The main part of the bit-slice ALU is a three-bit wide bit-slice which provides most of the ALU functionality. Before defining this bit-slice function, we define three decoding functions, generate, propagate, and mode, which control the bit-slice function. Each of the decoding functions takes only the operation code as an argument; they are all composed of random logic. These next five definitions should be skipped upon a first reading.

```
(DECODE-MODE OP-CODE)
 =
(B-AND (B-OR3 (BITN OP-CODE 1)
              (BITN OP-CODE 2)
              (BITN OP-CODE 3))
       (B-NOT (BITN OP-CODE 4)))

(DECODE-PROP OP-CODE)
 =
(LET ((OP1 (B-BUF (BITN OP-CODE 1)))
      (OP2 (B-BUF (BITN OP-CODE 2)))
      (OP3 (B-BUF (BITN OP-CODE 3)))
      (OP4 (B-BUF (BITN OP-CODE 4))))

  (BITV (B-OR (B-AND (B-NOT OP4) OP2)
              (B-AND3 (B-NOT OP2) OP3 OP4))
        (BITV (B-NAND3 OP1 (B-NOT OP2) OP3)
              (BITV (B-OR (B-AND3 OP2 (B-NOT OP3) (B-NOT OP4))
                          (B-AND3 OP4 (B-EQUV OP1 OP2)
                                  (B-XOR OP2 OP3)))
                    (BITV (B-AND OP3
                                 (B-OR (B-NOT OP4)
                                       (B-AND (B-NOT OP1) OP2)))
                          (BTM))))))
```

```
(DECODE-GEN OP-CODE)
 =
(LET ((OP1 (BITN OP-CODE 1))
      (OP2 (B-BUF (BITN OP-CODE 2)))
      (OP3 (B-BUF (BITN OP-CODE 3)))
      (OP4 (BITN OP-CODE 4)))

  (BITV (B-AND OP2 (B-NOT OP4))
        (BITV (B-AND4 OP1 (B-NOT OP2) OP3 (B-NOT OP4))
              (BITV (B-AND3 OP2 OP3 (B-NOT OP4))
                    (BTM)))))

(ALL-8 A B OP-CODE)
 =
(B-XOR3 (B-AND3 A B (BITN OP-CODE 1))
        (B-AND A (BITN OP-CODE 2))
        (B-AND B (BITN OP-CODE 3)))

(ALL-16 A B OP-CODE)
 =
(B-XOR4 (B-AND3 A B (BITN OP-CODE 1))
        (B-AND A (BITN OP-CODE 2))
        (B-AND B (BITN OP-CODE 3))
        (BITN OP-CODE 4))
```

DECODE-MODE produces a Boolean which is true for additions and subtractions, else false. **DECODE-PROP** and **DECODE-GEN** compute bit vectors which control the propagate and generate logic in the ALU. **ALL-8** and **ALL-16** define two combinational logic functions which compute one of either eight or sixteen logical functions.

Function **BV3-ALU-HELP** is the heart of the bit-slice ALU. This function recurs three bits at a time providing carry look-ahead over three bits.[3] If three bits do not remain in the input argument, then the result is computed with a size appropriate to the input.

```
(BV3-ALU-HELP C A B MODE OP-PROP OP-GEN)
 =
(LET ((A0 (BIT A))
      (B0 (BIT B))
      (A1 (BIT (VEC A)))
      (B1 (BIT (VEC B)))
      (A2 (BIT (VEC (VEC A))))
      (B2 (BIT (VEC (VEC B)))))

  (LET ((PROP0 (ALL-16 A0 B0 OP-PROP))
        (GEN0  (ALL-8  A0 B0 OP-GEN))
        (PROP1 (ALL-16 A1 B1 OP-PROP))
        (GEN1  (ALL-8  A1 B1 OP-GEN))
        (PROP2 (ALL-16 A2 B2 OP-PROP))
        (GEN2  (ALL-8  A2 B2 OP-GEN)))
```

[3]Why three bits? To dispell the notion that ALUs must be constructed in 2^N sized slices.

```
(LET ((C0 (B-AND MODE C))
      (C1 (B-AND MODE (B-OR GEN0 (B-AND C PROP0))))
      (C2 (B-AND MODE (B-OR3 GEN1
                             (B-AND PROP1 GEN0)
                             (B-AND3 PROP0 PROP1 C))))
      (C-OUT (B-OR4 GEN2
                    (B-AND PROP2 GEN1)
                    (B-AND3 PROP1 PROP2 GEN0)
                    (B-AND4 PROP0 PROP1 PROP2 C))))

  (LET ((F0 (B-XOR C0 (B-XOR PROP0 GEN0)))
        (F1 (B-XOR C1 (B-XOR PROP1 GEN1)))
        (F2 (B-XOR C2 (B-XOR PROP2 GEN2))))

    (COND
     ((BTMP A) (BITV C0 (BTM)))

     ((BTMP (VEC A))
      (BITV F0 (BITV C1 (BTM))))

     ((BTMP (VEC (VEC A)))
      (BITV F0 (BITV F1 (BITV C2 (BTM)))))

     (T (BITV F0
            (BITV F1
             (BITV F2
              (BV3-ALU-HELP C-OUT
                            (VEC (VEC (VEC A)))
                            (VEC (VEC (VEC B)))
                            (B-BUF MODE)
                            (BV-BUF OP-PROP)
                            (BV-BUF OP-GEN))))))))))
```

The above definition should be studied in sections. The first **LET** expression defines a set of names for the first three bits in each bit vector whether these bits exist or not. These bits are only referred to if it makes sense to do so. The second **LET** defines the propagate and generate components for each of the three bits in a slice. The third **LET** constructs the carry outs for each bit of the slice. These carry outs provide carry look-ahead for each slice. The fourth **LET** specifies the three output bits for one slice of the ALU. The body of this function then selects one of four results depending on the size of the input vector. For instance, if **(BTMP A)**, then just the carry-out is produced, and so on. It can be seen by inspection that **BV3-ALU-HELP** produces a bit vector result which has a size one greater than its **A** argument. This last bit is the carry out when **BV3-ALU-HELP** is performing addition and subtraction.

We now define the remaining three helper functions we require to build the bit-slice ALU. They assist in producing the ALU carry-out and overflow outputs and one provides help in providing the correct carry-in to the ALU. These functions were all composed by hand and are constructed from random logic. These next three definitions should be skipped upon a first reading.

```
(CARRY-IN-HELP C OP-CODE)
 =
(LET ((OP1 (B-BUF (BITN OP-CODE 1)))
      (OP2 (B-BUF (BITN OP-CODE 2)))
      (OP3 (B-BUF (BITN OP-CODE 3))))

   (B-OR (B-OR3 (B-AND (B-NOT OP2) (B-NOT OP3))
                (B-AND3 (B-NOT OP1) (B-NOT OP2) OP3)
                (B-AND3 OP1 OP2 OP3))
         (B-OR (B-AND (B-AND (B-NOT OP1) OP2)
                      (B-AND (B-NOT OP3) C))
               (B-AND (B-AND (B-NOT OP1) OP2)
                      (B-AND OP3 (B-NOT C))))))

(CARRY-OUT-HELP CIN A RESULT OP-CODE)
 =
(LET ((OP1 (B-BUF (BITN OP-CODE 1)))
      (OP2 (B-BUF (BITN OP-CODE 2)))
      (OP3 (B-BUF (BITN OP-CODE 3)))
      (OP4 (B-BUF (BITN OP-CODE 4))))

  (B-OR4 (B-AND4 (B-NOT OP4) (B-OR OP1 OP2) (B-NOT OP3) RESULT)
         (B-AND3 (B-NOT OP4) OP3 (B-NOT RESULT))
         (B-AND4 OP4 (B-NOT OP3) (B-XOR OP1 OP2) (BITN A 1))
         (B-AND5 (B-NOT OP1) (B-NOT OP2) (B-NOT OP3) OP4
                 (IF (BTMP A) CIN (BITN A 1)))))

(OVERFLOW-HELP TOP-BIT-RESULT TOP-A TOP-B OP-CODE)
 =
(LET ((OP1 (B-BUF (BITN OP-CODE 1)))
      (OP2 (B-BUF (BITN OP-CODE 2)))
      (OP3 (B-BUF (BITN OP-CODE 3)))
      (OP4 (B-BUF (BITN OP-CODE 4)))
      (AN  (B-BUF TOP-A))
      (BN  (B-BUF TOP-B))
      (TOP (B-BUF TOP-BIT-RESULT)))

  (B-OR4 (B-AND5 (B-NOT OP4) OP3 OP2
                 (B-XOR AN BN)
                 (B-XOR BN TOP))
         (B-AND4 (B-NOT OP4) OP3 (B-NOT OP2)
                 (B-OR (B-AND3 OP1 AN (B-NOT TOP))
                       (B-AND3 (B-NOT OP1) AN TOP)))
         (B-AND5 (B-NOT OP4) (B-NOT OP3) OP2
                 (B-EQUV AN BN)
                 (B-XOR BN TOP))
         (B-AND6 (B-NOT OP4) (B-NOT OP3) (B-NOT OP2)
                 OP1 (B-NOT AN) TOP)))
```

The bit-slice ALU is defined by the function **NEW-ALU**. This function just connects together the various functions defined above in a fashion suggested by Figure 2.

```
(NEW-ALU C A B OP-CODE)
 =
(LET ((OP (BV-BUF OP-CODE))
      (ASIZE (SIZE A)))

  (LET ((MODE (DECODE-MODE OP))
        (PROP (DECODE-PROP OP))
        (GEN  (DECODE-GEN  OP))
        (AX   (BV-SHIFT-OR-BUF C A OP))
        (CX   (CARRY-IN-HELP C OP)))

    (LET ((BV3-ALU (BV3-ALU-HELP CX AX B MODE PROP GEN)))

      (BV-CV (TRUNC BV3-ALU ASIZE)
             (CARRY-OUT-HELP C A (BITN BV3-ALU (ADD1 ASIZE)) OP)
             (OVERFLOW-HELP (BITN BV3-ALU ASIZE)
                            (BITN A ASIZE)
                            (BITN B ASIZE)
                            OP)))))
```

This completes the definition of the bit-slice ALU.

5. The ALU Correspondence Proof

The proof of correspondence of the bit-slice and FM8502 ALUs is by no means straightforward. We construct a graduated sequence of lemmas which leads the theorem prover to the proof. There were several major steps in this development which we outline here.

The Boyer-Moore theorem prover is used by entering definitions and proof requests. Definitions, if accepted, are stored in its internal database and are used during proofs as needed. Proof requests cause the theorem prover to attempt to establish the validity of a conjecture. Proved conjectures, which we call lemmas, are also stored in the theorem prover database. Previously defined functions and proved lemmas may be used in a current theorem prover request.

Definitions and lemmas for the correspondence proof are entered into the Boyer-Moore theorem prover with a script containing all the necessary commands. The ordering of the commands is important, e.g., a definition cannot be used before it is defined.

The first part of the correspondence proof script is concerned with defining Booleans and bit vectors. We then define several functions which manipulate bit vectors (e.g., **TRUNC**, **BITN**, **SIZE**, etc.) and prove often needed properties of these definitions. We next define our primitive hardware functions (e.g., **B-NOT**, **B-AND**, etc.) and their vector versions. The sizes of these vector hardware functions are noted with a lemma.

We proceed by defining the adder and subtractor functions; these were taken from the original FM8501 proof script. We define the previously described selectors: **BV-IF** and **BV4-IF**. We then define the **BV-CV-IF** selector. These definitions are all proved to implement selectors. The FM8501 ALU is defined next.

The bit-slice ALU definition proceeds in exactly the same fashion as described in the last section, except we prove some lemmas about **BV-SHIFT-OR-BUF** just after its definition.

We are now ready to begin the correspondence proof. This proceeds in two basic steps: the inductive proofs that the bit-slice ALU help function **BV-ALU-HELP** can provide the required logical and arithmetic operations, and the gluing together of these inductive proofs to complete the proof. Let us examine one of the inductive proof statements; below we state that **BV-ALU-HELP**, under certain conditions, operates just like the vector-and function **BV-AND**.

```
(IMPLIES
 (AND (BITVP A)
      (BITVP B)
      (EQUAL MODE F)
      (EQUAL OP-PROP (BITV T (BITV F (BITV F (BITV F (BTM))))))
      (EQUAL OP-GEN  (BITV F (BITV F (BITV F (BTM))))))
 (EQUAL (BV3-ALU-HELP C A B MODE OP-PROP OP-GEN)
        (V-APPEND (BV-AND A B) (BITV F (BTM)))))
```

Notice we append a false Boolean value onto the result of the **BV-AND** function because **BV3-ALU-HELP** produced a vector one bit longer than its input.

After proving lemmas describing the operation of **BV3-ALU-HELP** for the various functions our bit-slice ALU provides, we prove the final theorem below. This final theorem is obtained by examining the cases of the operation code and using the lemmas described just above.

```
(IMPLIES (AND (BOOLP C)
              (BV2P A B)
              (BITVP OP-CODE))

         (EQUAL (NEW-ALU C A B OP-CODE)
                (BV-ALU-CV A B C OP-CODE)))
```

The Boyer-Moore theorem prover requires 15 minutes to process the ALU correspondence proof script on a Sun 3/280. This time includes defining Booleans and bit vectors, processing the ALU definitions, and proving their correspondence. We then may use these definitions in other proofs. In fact, the FM8501 ALU has been verified to have abstract properties with respect to Boolean bit vectors, natural numbers, and integers. In turn, these abstract properties are used in the verification of the Piton assembler [Moore 88] for the FM8502.

6. ALU Expansions

The FM8501 and bit-slice ALUs specify very different implementations. The FM8501 ALU design is overly simple for actual hardware use. The bit-slice ALU design specifies an implementation which is comparable in gate count to actual working implementations. In this section we present comparisons of gate counts, fanouts, and delays for each ALU.

Our ALU implementation specifications are blueprints for ALUs of any size. We compare our ALUs by expanding our specifications into a number of fixed-sized ALUs. Gates are required even when the bit-vector portion of the result of the ALUs is empty; these gates compute the carry out and overflow results. The zero-bit, bit-slice ALU requires more gates than the zero-bit, FM8501 ALU because of the greater amount of decoding logic used in the bit-slice ALU.

ALU Gate Graph Comparison						
ALU Size in Bits	FM8501 Gate Count	FM8501 Fanout	FM8501 Delay	Bit-slice Gate Count	Bit-slice Fanout	Bit-slice Delay
0-bits	37	5	10	64	4	11
1-bit	205	39	13	137	7	19
2-bits	345	48	16	157	7	21

ALU Size in Bits	FM8501 Gate Count	FM8501 Fanout	FM8501 Delay	Bit-slice Gate Count	Bit-slice Fanout	Bit-slice Delay
4-bits	665	80	22	205	7	21
8-bits	1305	144	34	295	7	25
16-bits	2585	272	58	483	7	29
32-bits	5145	528	106	851	7	41
64-bits	10265	1040	202	1595	7	61
128-bits	20505	2064	394	3075	7	105

We can see how much better the bit-slice ALU is than the FM8501 ALU. The fanouts in the FM8501 ALU are unrealistic and the delay is too long. The bit-slice ALU benefits from using three and four input gate primitives, whereas the FM8501 ALU only uses two input gate primitives. This does not affect the gate count that much, for instance, the 32-bit version of the bit-slice ALU has 1020 two-input primitive gates instead of the 851 reported in the table above. However, restricting the implementation to one and two input gate primitives affects the delay more dramatically by increasing it from 41 to 64.

The widest primitive gates we used have four inputs. The three-bit wide ALU slice takes best advantage of these primitives as can be observed in the computation of the carry outs in the function **`BV3-ALU-HELP`**. With wider primitives it is possible to make faster and faster ALUs by wider slices.

The expansion process just unfolds of definitions, notes common subexpressions, and applys applicable rewrite rules. The 128-bit, bit-slice ALU gate graph takes 100 seconds to generate.

7. Conclusions

The verification of the bit-slice ALU with respect to the FM8501 ALU has been accomplished by using a general-purpose mechanical theorem prover. These ALU specifications demonstrate the use of a functional language as a combinational logic design language. The verification of these ALU specifications represents an exhaustive comparison of Boolean functions without using exhaustive techniques.

The FM8501 ALU specification is too inefficient to be useful as a specification for real hardware; however, it is a useful intermediate specification step on the way to verifying arithmetic properties of the bit-slice ALU. The FM8501 ALU is composed of simple internal functions connected together with selectors. This arrangement provided a straightforward path to verify the arithmetic properties of the FM8501 ALU. Each internal function is studied independently, and its properties identified and verified. Using these properties and our knowledge about selectors, we are able to compose an ALU with known mathematical properties. We still know of no other ALU whose operations are completely verified with respect to Peano and integer arithmetic.

The bit-slice ALU specification has a gate structure comparable in gate count and gate delay to a serially interconnected set of 74181s composed with a right-shift unit. The bit-slice ALU does not have the simple structure of the FM8501 ALU, and thus verifying the arithmetic properties of the FM8502 ALU directly is more difficult than verifying the FM8501 ALU arithmetic properties. The purpose of verifying the bit-slice ALU with respect to the FM8501 ALU is to ensure that the bit-slice ALU has the arithmetic properties we desire.

The bit-slice specification was developed and verified in a hierarchical fashion. The ALU is composed of several modules as pictured in Figure 2. Some of these modules are composed of sub-modules. For instance, the **`Shift or Buffer`** module is composed of several sub-modules: a selector, a shifter, and some decoding logic. We performed the verification hierarchically also. In the case of the **`Shift or Buffer`** module we verified a selector lemma and then used this lemma in the verification of the module. Later, the **`Shift or Buffer`** module lemmas were used in the verification of the entire ALU.

We were able to design and verify the bit-slice ALU in several weeks time. Small changes (of which we made many) often only changed a part of the proof, and when we made errors (not quite so many) we found our problems quickly. In fact, the Boyer-Moore theorem prover output was instrumental in the location of errors, in that it often became "stuck" right at the error in question.

The use of a general-purpose logic allows more abstract circuit specifications than Boolean decision procedures allow. For instance, when specifying addition we refer to numbers; these simply do not exist in Boolean decision procedures. Here we did not make use of this generality, but instead demonstrated the Boolean equivalence of two large functions.

We believe recursion is an underutilized method for dealing with large circuits. Hardware by its very nature is quite repetitive, and recursion captures this regularity. Recursively defined hardware can be verified with induction; this ability is lacking from Boolean decision procedures. We expect our recursive techniques to extend directly to multipliers and other more complicated circuits.

References

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

[Bryant 86] Randal E. Bryant.
Graph-Based Algorithms for Boolean Function Manipulation.
IEEE Transactions on Computers C-35(8):677--691, August, 1986.

[Hunt 85] Warren A. Hunt, Jr.
FM8501: A Verified Microprocessor.
Technical Report ICSCA-CMP-47, University of Texas at Austin, 1985.

[Hunt 87] Warren 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.

[Hunt 89] Warren A. Hunt, Jr.
Microprocessor Design Verification.
Journal of Automated Reasoning (to appear), 1989.

[Moore 88] J S. Moore.
Piton: A Verified Assembly Level Language.
Technical Report 22, Computational Logic, Inc., 1717 West Sixth Street, Suite 290 Austin, TX 78703, 1988.

[Steele 84] Guy L. Steele Jr.
Common LISP: The Language.
Digital Press, 1984.

Table of Contents

List of Figures

List of Tables