The Formalization of a Simple Hardware Description Language

Bishop C. Brock
Warren A. Hunt, Jr.

Technical Report 52
November 1989

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

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

Abstract. A hierarchical, occurrence-oriented, combinational hardware description language has been formalized using the Boyer-Moore logic. Instead of representing circuits as formulas of a particular logic, combinational circuits are represented by list constants in the Boyer-Moore logic. A good-circuit predicate recognizes well-formed circuit descriptions; an interpreter provides the semantics of the language. This approach allows the direct verification of circuit specifications, as well as allowing the verification of circuit generating functions. A circuit generating function for a family of ALUs has been verified using these techniques.

1. Introduction

The formalization of a hierarchical, occurrence-oriented, combinational hardware description language (HDL) has been employed to prove the correctness of functions which generate circuits. This formalization was carried out with the Boyer-Moore logic and its associated mechanical theorem prover [Boyer & Moore 88]. HDL statements are formalized as list constants in the Boyer-Moore logic, and the proofs of correctness of HDL circuits are carried out using the Boyer-Moore theorem prover.

Previously we used Boyer-Moore logic expressions [Hunt 89a, Hunt 89b] to represent and verify circuits; however, this representation provided inadequate information for conventional CAD systems. Our formalization of a hierarchical, occurrence-oriented HDL using constants provides a formal representation for circuits which is closely related to HDLs currently in use in industry. With our new formalization, a *circuit box* is specified by listing its inputs and outputs, and its internal circuitry. Internal circuitry is specified as interconnected primitive gates and/or other boxes, allowing circuits to be specified and verified hierarchically. Our simple HDL model only recognizes combinational primitives. Also, circuitry within a box may not be circular, nor may box compositions introduce circularities.

We make the notion of a circuit box precise with a predicate, written in the Boyer-Moore logic, which we map over circuit descriptions. We also use the Boyer-Moore logic to define several interpreters which evaluate circuit properties. For proving circuit correctness we use an interpreter which computes a value for a circuit given its inputs. We can also compute or verify the delay and loading of a circuit or class of circuits.

The use of general-purpose theorem proving systems to guarantee the correctness of circuit designs is a recent advance [Gordon 83, Viper 89, Hunt 85]. With other hardware verification efforts that we are aware of, circuits are represented as formulas in a particular logic, theorems are proved about these representations, and then the logic formulas are translated into an language suitable for actual design layout. (In other cases, circuits are originally described in a HDL, but for verification purposes they are translated into a formal logic.) These kinds of translations are impossible to guarantee because of the generally informal nature of CAD languages.

Our approach, however, formalizes conventional hardware design environments. Circuits are described by list constants which capture the abstract syntax of a generic HDL. Our good-circuit predicate is analogous to a netlist compiler, ensuring that the netlist is syntactically valid. Our evaluation function plays the role of a simulator, and more. Since our evaluator is embedded in the Boyer-Moore logic, we are able to prove abstract properties of our circuits with respect to the evaluation function, as well as simply simulate circuits on fixed inputs. Similar to the way that hardware engineers can write programs to create the text of circuit descriptions, we have written functions that generate circuits in our abstract syntax, and furthermore have proven that the circuits generated by these functions are correct with respect to their abstract specifications.

In this note we explore our HDL formalization by considering the verification of a circuit generating

function for a family of ALUs. Before describing our ALU proof, we introduce the idea of hardware verification and describe the formalization of our HDL. We then consider the verification of several simple circuits and conclude with our ALU verification.

2. Our Approach

Hardware verification is the application of formal methods to the task of specifying and verifying the operation of digital computing.[1] Typically, digital hardware is designed from natural language specifications. With our approach, circuits are specified, both abstractly and concretely, as mathematical formulas. Formally specified circuits and designs allow us to rigorously demonstrate that designs meet their specifications by using formal proof techniques. In this section, we argue the merits of the approach to hardware verification outlined in this note.

Formal techniques are being employed by a number of groups (for example [Brown 89, Viper 89, Gordon 85, Johnson 89, Sheeran 84]) to aid in the specification, design, and verification of computer hardware. Each group has a formal logic which is used to model hardware. Typically, functions or predicates are defined in the logic of choice, and these functions or predicates are considered to represent either collections of primitive objects or primitive objects themselves. In other words, functions or predicates of some logic are "overloaded" in the sense that they have both a formal meaning in the defining logic, and they are used to represent computer hardware.

For example, consider the one-bit selector circuit schematic in Figure 1. The typical approach taken to

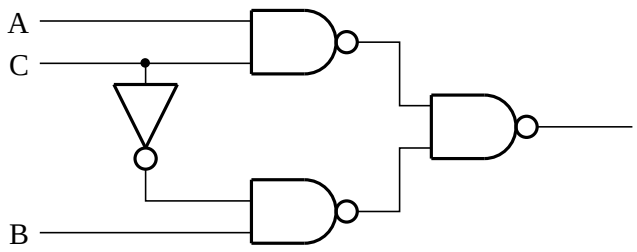

Figure 1: One-bit Selector Circuit

represent this circuit formally is to introduce a function or predicate describing the operation of each gate (where in turn, each gate is described as either primitives of the logic being used and/or other functions or predicates.) Using the Boyer-Moore logic we can formalize the one-bit selector circuit with the **B-IF** function.

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

That is, we introduce the function **B-IF**, which is defined in terms of the functions **B-NAND** and **B-NOT**. For this to be possible in the Boyer-Moore logic, it is necessary for **B-AND** and **B-NOT** to have been previously defined, and in this case, these two functions were defined with primitives of the Boyer-Moore logic. Additionally, the functions **B-NAND** and **B-NOT** are overloaded in the sense that we think of these functions as representing hardware gates.

[1] This is not to say that it is impossible to verify analogue circuits; we only consider digital circuits.

We find overloading functions in the Boyer-Moore logic to be imprecise when considering if a particular circuit description represents a circuit which can be physically realized. In general it is impossible for us to formally decide (inside the logic) whether an arbitrary function was defined in such a way that it can be completely represented with primitive hardware functions. Another problem we suffer by using functions in the Boyer-Moore logic is that fanout cannot be explicitly expressed. By inspecting the definition of **B-IF** we can see that **C** appears twice, suggesting that **C** fans out twice; however, we still have no explicit means of checking this. Modeling circuits with predicates allows the explicit specification of fanout, but yields unexecutable specifications, and introduces a host of other problems concerning the realizability of the specified hardware.

We have several goals in the formalization of an HDL: its structure should be similar to commercial HDL's, it should be executable, and it should provide a basis for verified synthesis. In the past, we have found it more difficult to precisely specify a circuit for manufacturing than to check whether a specification represents a circuit. Our HDL formalization now allows us to explicitly specify circuit interconnections. The ability to explicitly manipulate circuit expressions cannot be overemphasized. The structure of our HDL allows simulation of our circuit specifications without having to transform them into some other form. Very often circuit designs are refined from specifications and then optimized. Since our circuit descriptions are simply constants in the Boyer-Moore logic, we are able to synthesize and manipulate circuit descriptions with provably correct procedures.

3. The Boyer-Moore Logic

We use the Boyer-Moore logic as a hardware description language and also as a hardware specification language. Here we introduce the Boyer-Moore logic and give several examples of its use.

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 propositional logic and linear arithmetic, a simplifier, and a rewriter. The Boyer-Moore theorem prover also contains procedures for automatically performing structural inductions.

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.

Consider that we wish to represent a bit vector as a list which contains only Boolean elements. We formalize this notion within the Boyer-Moore logic by introducing the functions **BOOLP** and **BVP**. We write function definitions with the symbol "=", while theorems are presented without the "=" symbol.

```
(BOOLP X) = (OR (EQUAL X T) (EQUAL X F))

(BVP X) = (IF (NOT (LISTP X))
              (EQUAL X NIL)
              (AND (BOOLP (CAR X))
                   (BVP (CDR X))))

(BITN N LIST) = (IF (ZEROP N)
                    (IF (LISTP LIST)
                        (IF (CAR LIST) T F)
                        F)
                    (BITN (SUB1 N) (CDR LIST)))
```

BOOLP is just an abbreviation, while **BVP** is a recursive function. For a recursive function to be acceptable, it must be possible to prove that it terminates. Pairs are constructed with **CONS**; **CAR** selects the first element of a pair and **CDR** the second. If a bit vector is empty, **BVP** insists that it equal **NIL**. For a list, the first element must be Boolean and the remainder of the list must be a bit vector. We later use this formalization of Booleans and bit vectors. **BITN** selects bit **N** from **LIST** if **N** is less than the length of **LIST**; otherwise **BITN** returns **F**.

Using the function **APPEND**, we can prove that appending two bit vectors together produces a bit vector.

```
(APPEND X Y) = (IF (NOT (LISTP X))
                   Y
                   (CONS (APPEND (CAR X) Y)))

(IMPLIES (AND (BVP X)
              (BVP Y))
         (BVP (APPEND X Y)))
```

The proof is by induction on **X**, and the Boyer-Moore theorem prover can automatically perform this proof.

To represent our hardware circuit boxes we are using Boyer-Moore logic constants. Constants are written using the Lisp quote notation. The following statements are theorems.

```
(EQUAL (CAR (CONS X Y) X))
(EQUAL (CDR (CONS X Y) Y))

(EQUAL (LISTP (CONS X Y)) T)

(EQUAL '(A B C ... X) (CONS 'A '(B C ... X)))

(EQUAL (CAR '(A B C) 'A))
(EQUAL (CDR '(A B C) '(B C)))

(EQUAL (LIST A B C ... X) (CONS A (LIST B C ... X)))
```

'(A B C) is a list of three literal atoms. We use (nested) lists of this kind to represent our circuit descriptions. We access components of circuit description with **CAR** and **CDR**. We abbreviate nests of **CAR** and **CDR**; for example, we abbreviate **(CAR (CDR (CDR X)))** by **(CADDR X)**.

Below are several functions which we will use throughout the remainder of this note.

```
(NLISTP X) = (NOT (LISTP X))

(ASSOC X ALIST) = (IF (NLISTP ALIST)
                      F
                      (IF (EQUAL X (CAAR ALLIST))
                          (CAR ALIST)
                          (ASSOC X (CDR ALIST))))
```

```
(DISJOINT L1 L2) = (IF (NLISTP L1)
                       T
                       (AND (NOT (MEMBER (CAR L1) L2))
                            (DISJOINT (CDR L1) L2)))

(DUPLICATES? L) =  (IF (NLISTP L)
                       F
                       (OR (MEMBER (CAR L) (CDR L))
                           (DUPLICATES? (CDR L))))

(LENGTH X) = (IF (NLISTP X) 0 (ADD1 (LENGTH (CDR X))))

(MEMBER X LIST) = (IF (NLISTP LIST)
                      F
                      (IF (EQUAL X (CAR LIST))
                          T
                          (MEMBER X (CDR LIST))))

(PAIRLIST L1 L2) = (IF (NLISTP L1)
                       NIL
                       (CONS (CONS (CAR L1) (CAR L2))
                             (PAIRLIST (CDR L1) (CDR L2)))))

(PROPERP X) = (IF (NLISTP X) (EQUAL X NIL) (PROPERP (CDR X)))

(SUBSET L1 L2) = (IF (NLISTP L1)
                     T
                     (AND (MEMBER (CAR L1) L2)
                          (SUBSET (CDR L1) L2)))
```

4. Some Example Circuit Boxes

To give the flavor of our HDL formalization, we present some example circuits along with their HDL specifications. Circuit boxes are specified with Boyer-Moore logic list constants. A well-formed circuit box contains four items: a name, a list of inputs, a list of outputs, and a circuit description body. The names of the inputs and outputs must be distinct. The body is a list of wiring instructions.

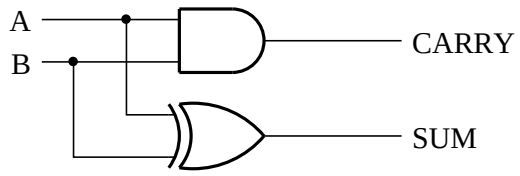

Figure 2: Half-Adder Circuit

Circuit boxes for a half-adder and a full-adder are below. Certain circuit boxes are considered primitive, for example **B-XOR**, **B-AND**, and **B-OR**. The **HALF-ADDER** circuit box contains just two primitives, and this circuit is pictured in Figure 2.

```
'(HALF-ADDER (A B) (SUM CARRY) (((SUM)   (B-XOR A B))
                                ((CARRY) (B-AND A B))))
```

```
'(FULL-ADDER (A B C) (SUM CARRY)
             (((SUM1 CARRY1) (HALF-ADDER A B))
              ((SUM CARRY2)  (HALF-ADDER SUM1 C))
              ((CARRY)       (B-OR CARRY1 CARRY2))))
```

The **FULL-ADDER** circuit box makes two references to **HALF-ADDER**, thus to build a **FULL-ADDER** circuit requires two half-adders and one **B-OR** primitive, as pictured in Figure 3. A complete circuit description is represented as a list of circuit boxes, which we call a *boxlist*.

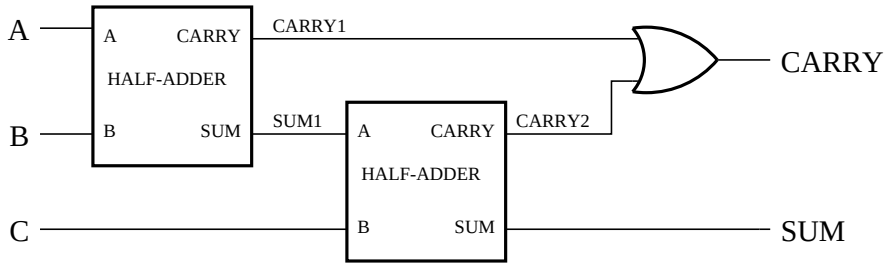

Figure 3: Full-Adder Circuit

5. Boxlist Syntax

Here we present the specification for well-formed circuit boxes and boxlists. A circuit box may refer to other circuit boxes; every circuit box reference in a well-formed boxlist either refers to other circuit boxes in the boxlist or to primitives. The form of a boxlist prevents circuit boxes from referring to themselves. The class of circuits which can be expressed with a well-formed boxlist is just combinational logic without feedback.

The syntax of our HDL is formally specified by the function **BOXLIST-OKP**. A boxlist which is **NIL** is well-formed; otherwise, the **(CAR BOXLIST)** must be a syntactically well-formed circuit box in the context of **(CDR BOXLIST)**, which also must be well-formed. The conditions on **(CAR BOXLIST)** include: a unique name, disjoint input and output names, and a well-formed body.

```
(BOXLIST-OKP BOXLIST)
 =
(IF (NLISTP BOXLIST)
    (EQUAL BOXLIST NIL)
    (LET ((BOX (CAR BOXLIST)))
      (AND (LISTP BOX)
           (LISTP (CDR BOX))
           (LISTP (CDDR BOX))
           (LISTP (CDDDR BOX))
           (EQUAL (CDDDDR BOX) NIL)
           (LET ((NAME    (CAR BOX))
                 (INPUTS  (CADR BOX))
                 (OUTPUTS (CADDR BOX))
                 (BODY    (CADDDR BOX)))
```

```
            (AND (NOT (ASSOC NAME (CDR BOXLIST)))
                 (PROPERP INPUTS)
                 (NOT (DUPLICATES? INPUTS))
                 (PROPERP OUTPUTS)
                 (NOT (DUPLICATES? OUTPUTS))
                 (DISJOINT INPUTS OUTPUTS)
                 (BODY-OKP BODY INPUTS OUTPUTS (CDR BOXLIST))
                 (BOXLIST-OKP (CDR BOXLIST)))))))
```

A well-formed body is specified by the function **BODY-OKP**. A body contains a list of occurrences, where each occurrence is a list of two elements: an output list and a circuit box reference. Each occurrence in a body satisfies the following conditions: the circuit box reference is either a primitive reference or a reference to another circuit box, the input arity of a circuit box reference must match the box being referenced, a value for each input must exist, the output arity of a circuit box reference must match the occurrence output list, each occurrence output list cannot contain duplicates, and all occurrence output lists must be disjoint. Furthermore, every output must be set exactly once.

```
(BODY-OKP BODY SIGNALS OUTPUTS BOXLIST)
 =
(IF (NLISTP BODY)
    (AND (EQUAL BODY NIL)
         (SUBSET OUTPUTS SIGNALS))
    (LET ((OCCURRENCE (CAR BODY)))
      (AND
       (LISTP OCCURRENCE)
       (LISTP (CDR OCCURRENCE))
       (EQUAL (CDDR OCCURRENCE) NIL)
       (LET ((LHS (CAR OCCURRENCE))
             (RHS (CADR OCCURRENCE)))
         (AND
          (PROPERP LHS)
          (NOT (DUPLICATES? LHS))
          (DISJOINT LHS SIGNALS)
          (LISTP RHS)
          (PROPERP RHS)
          (LET ((MODULE-NAME (CAR RHS))
                (MODULE-ARGS (CDR RHS)))
            (LET ((PRIMP (PRIMP MODULE-NAME)))
             (AND
             (SUBSET MODULE-ARGS SIGNALS)
             (IF PRIMP
                 (AND (EQUAL (CAR PRIMP) (LENGTH MODULE-ARGS))
                      (EQUAL (CDR PRIMP) (LENGTH LHS)))
                 (LET ((SUBMODULE (ASSOC MODULE-NAME BOXLIST)))
                   (LET ((SUBMODULE-INPUTS  (CADR SUBMODULE))
                         (SUBMODULE-OUTPUTS (CADDR SUBMODULE)))
                     (AND SUBMODULE
                          (EQUAL (LENGTH MODULE-ARGS)
                            (LENGTH SUBMODULE-INPUTS))
                          (EQUAL (LENGTH LHS)
                                 (LENGTH SUBMODULE-OUTPUTS))))))
             (BODY-OKP (CDR BODY) (APPEND LHS SIGNALS)
                       OUTPUTS BOXLIST)))))))))
```

PRIMP specifies the output and input arity for each primitive.

```
(PRIMP FN)
 =
(CASE FN
      (B-BUF   (CONS 1 1))    (B-NOT   (CONS 1 1))
      (B-NAND  (CONS 2 1))    (B-NAND3 (CONS 3 1))
      (B-NAND4 (CONS 4 1))    (B-OR    (CONS 2 1))
      (B-OR3   (CONS 3 1))    (B-OR4   (CONS 4 1))
      (B-EQUV  (CONS 2 1))    (B-XOR   (CONS 2 1))
      (B-AND   (CONS 2 1))    (B-AND3  (CONS 3 1))
      (B-AND4  (CONS 4 1))    (B-NOR   (CONS 2 1))
      (B-NOR3  (CONS 3 1))    (B-NOR4  (CONS 4 1))
      (OTHERWISE F))
```

6. The Logical Hardware Interpreter

Since our HDL is written as constants, we have defined interpreters to give various meanings to our circuit boxes and boxlists. Here we consider the definition of the function **HEVAL**, which interprets references to a circuit box as Boolean functions.

HEVAL uses an association list to assign values to (signal) names. **EVAL-NAME** retrieves the value assigned to **NAME** in **ALIST** and coerces the value to a Boolean. **COLLECT-EVAL-NAME** retrieves a list of values.

```
(EVAL-NAME NAME ALIST)
 =
(IF (NLISTP ALIST)
    F
    (IF (AND (LISTP (CAR ALIST)) (EQUAL NAME (CAAR ALIST)))
        (BOOLFIX (CDAR ALIST))
        (EVAL-NAME NAME (CDR ALIST))))

(COLLECT-EVAL-NAME ARGS ALIST)
 =
(IF (NLISTP ARGS)
    NIL
    (CONS (EVAL-NAME (CAR ARGS) ALIST)
          (COLLECT-EVAL-NAME (CDR ARGS) ALIST)))
```

To interpret a circuit box body, **FLAG** is set to **F**; otherwise, **HEVAL** interprets **FORM** as a circuit box reference. A circuit box is interpreted in two steps: values for the arguments are retrieved from **ALIST** and the circuit box reference is applied to the argument values. **(HAPPLY FN ARGS)** provides a value for each primitive **FN** recognized by **PRIMP** given **ARGS**. For example, **(HAPPLY 'B-NOT (LIST T))** is **(LIST F)**. Otherwise, the circuit box reference is retrieved from **BOXLIST** and its body is interpreted by **HEVAL**.

```
(HEVAL FLAG FORM ALIST BOXLIST)
 =
(IF FLAG
    (LET ((FN   (CAR FORM))
          (ARGS (CDR FORM)))
      (LET ((ACTUALS (COLLECT-EVAL-NAME ARGS ALIST)))
        (IF (PRIMP FN)
            (HAPPLY FN ACTUALS)
            (LET ((BOX (ASSOC FN BOXLIST)))
              (IF BOX
                  (LET ((INPUTS  (CADR BOX))
                        (OUTPUTS (CADDR BOX))
                        (BODY    (CADDDR BOX)))
                    (COLLECT-EVAL-NAME
                     OUTPUTS
                     (HEVAL F BODY (PAIRLIST INPUTS ACTUALS)
                            (CDR BOXLIST))))
                  F)))))

(IF (LISTP FORM)
    (LET ((OCCURRENCE (CAR FORM)))
     (LET ((LHS (CAR OCCURRENCE))
           (RHS (CADR OCCURRENCE)))
        (HEVAL F
               (CDR FORM)
         (APPEND (PAIRLIST LHS (HEVAL T RHS ALIST BOXLIST))
                 ALIST)
                BOXLIST)))
   ALIST))
```

To illustrate the workings of **HEVAL** we consider the interpretation of **'(HALF-ADDER A B)** with the following environment.

```
BOXLIST = (LIST '(HALF-ADDER (A B) (SUM CARRY)
                             (((SUM)   (B-XOR A B))
                              ((CARRY) (B-AND A B)))))

ALIST = (LIST (CONS 'X F) (CONS 'Y T))
```

We initiate the interpretation of **HALF-ADDER** by the following invocation of **HEVAL**, which finally simplifies to **(LIST T F)**.

```
(HEVAL T '(HALF-ADDER X Y) ALIST BOXLIST)
 =
(COLLECT-EVAL-NAME '(SUM CARRY)
                   (HEVAL F '(((SUM) (B-XOR A B))
                              ((CARRY) (B-AND A B)))
                            (LIST (CONS 'A F) (CONS 'B T)) NIL))
 =
(COLLECT-EVAL-NAME '(SUM CARRY)
                   (HEVAL F '(((CARRY) (B-AND A B)))
                            (LIST (CONS 'SUM T)
                                  (CONS 'A F) (CONS 'B T)) NIL))
 =
(COLLECT-EVAL-NAME '(SUM CARRY)
                   (HEVAL F '()
                            (LIST (CONS 'CARRY F) (CONS 'SUM T)
                                  (CONS 'A F) (CONS 'B T))
                            NIL))
 =
(COLLECT-EVAL-NAME '(SUM CARRY)
                   (LIST (CONS 'CARRY F) (CONS 'SUM T)
                         (CONS 'A F) (CONS 'B T)))
 =
(LIST T F)
```

We have also defined interpreters like **HEVAL** which compute the delay, primitive count, and loading of circuit box references.

7. A Simple Circuit Generator

A consequence of using list constants to specify hardware circuits is the ability to verify functions which create circuit descriptions. We demonstrate this ability with a simple example, the verification of a function which creates descriptions of *n*-bit adder circuits. This example is simple enough that we can completely describe it; the ALU generator verification discussed later is far too large. This adder example is often used in descriptions of formal hardware verification methodologies; thus the interested reader can directly compare this treatment with other approaches [Gordon 86, Goguen 89, Hunt 85, Pace 89].

Our abstract specifications are given by functions in the Boyer-Moore logic. The function **V-ADDER** is a specification for bit-vector addition.

```
(V-ADDER C A B)
  =
(IF (NLISTP A)
    (CONS (IF C T F) NIL)
    (CONS (XOR C (XOR (CAR A) (CAR B)))
          (V-ADDER (OR (AND (CAR A) (CAR B))
                       (OR (AND (CAR A) C)
                           (AND (CAR B) C)))
                   (CDR A)
                   (CDR B))))
```

The function **V-TO-NAT** provides an interpretation of bit-vectors as natural numbers. Using this interpretation, it is straightforward to prove that **V-ADDER** can perform natural number addition.

```
(V-TO-NAT X) = (IF (NLISTP X)
                   0
                   (PLUS (IF (CAR X) 1 0)
                         (TIMES 2 (V-TO-NAT (CDR X)))))
```

```
(IMPLIES (AND (BVP A) (BVP B)
              (EQUAL (LENGTH A) (LENGTH B)))
         (EQUAL (V-TO-NAT (V-ADDER C A B))
                (PLUS (IF C 1 0)
                      (V-TO-NAT A)
                      (V-TO-NAT B))))
```

Given the specification of bit-vector addition, we now turn to the creation of an HDL circuit which meets that specification. We proceed by defining a set of functions which generate a complete circuit description for an *n*-bit adder.

```
(GENERATE-NAMES SEED N)
  =
(IF (ZEROP N)
    NIL
    (CONS (CONS SEED N) (GENERATE-NAMES SEED (SUB1 N))))

(V-ADDER-BODY N)
  =
(IF (ZEROP N)
    NIL
    (CONS (LIST (LIST (CONS 'SUM N) (CONS 'CARRY N))
                (LIST 'FULL-ADDER (CONS 'A N) (CONS 'B N)
                      (CONS 'CARRY (ADD1 N))))
          (V-ADDER-BODY (SUB1 N))))

(V-ADDER* N)
  =
(LIST (CONS 'V-ADDER N)                          ; Name
      (CONS (CONS 'CARRY (ADD1 N))               ; Inputs
            (APPEND (GENERATE-NAMES 'A N)
                    (GENERATE-NAMES 'B N)))
      (APPEND (GENERATE-NAMES 'SUM N)            ; Outputs
              (LIST (CONS 'CARRY 1)))
      (V-ADDER-BODY N))                          ; Body
```

Since our simple HDL does not include a syntax for bit-vectors, we use the function **(GENERATE-NAMES SEED N)** to create lists of unique names. **(V-ADDER-BODY N)** generates a list constant representing the body of an *n*-bit adder, designed to be used in the signal name environment provided by **V-ADDER***. The adder is composed of interconnected **FULL-ADDER** modules using a ripple-carry scheme. **V-ADDER*** provides a circuit box for an *n*-bit ripple-carry adder. The following example should help clarify the circuit discriptions specified by **V-ADDER***.

```
(V-ADDER* 2)
  =
'((V-ADDER . 2)                                   ; Name
  ((CARRY . 3) (A . 2) (A . 1) (B . 2) (B . 1))   ; Inputs
  ((SUM . 2) (SUM . 1) (CARRY . 1))               ; Outputs
  ((((SUM . 2) (CARRY . 2))                       ; Body
    (FULL-ADDER (A . 2) (B . 2) (CARRY . 3)))
   (((SUM . 1) (CARRY . 1))
    (FULL-ADDER (A . 1) (B . 1) (CARRY . 2)))))
```

V-ADDER-BOXLIST provides a complete boxlist for our ripple carry adder by including circuit boxes for **FULL-ADDER** and **HALF-ADDER**.

```
(V-ADDER-BOXLIST N)
  =
(LIST (V-ADDER* N)
      '(FULL-ADDER (A B C) (SUM CARRY)
                   (((SUM1 CARRY1) (HALF-ADDER A B))
                    ((SUM CARRY2)  (HALF-ADDER SUM1 C))
                    ((CARRY)       (B-OR CARRY1 CARRY2))))
      '(HALF-ADDER (A B) (SUM CARRY)
                   (((SUM)   (B-XOR A B))
                    ((CARRY) (B-AND A B)))))
```

One way to state the correctness of the circuit produced by **V-ADDER-BOXLIST** is given below.

```
(IMPLIES (AND (EQUAL (LENGTH A) N)
              (EQUAL (LENGTH B) N))
         (EQUAL (HEVAL T (CONS (CONS 'V-ADDER N)
                               (CONS C (APPEND A B)))
                       ALIST (V-ADDER-BOXLIST N))
                (V-ADDER (EVAL-NAME C ALIST)
                         (COLLECT-EVAL-NAME A ALIST)
                         (COLLECT-EVAL-NAME B ALIST))))
```

This is a general lemma which specifies that the **HEVAL** interpretation of a reference to the **V-ADDER** box in **(V-ADDER-BOXLIST N)** is the same as **V-ADDER** applied to the values in the **ALIST** bound to the symbolic inputs. A more concrete consequence of the lemma just above is presented below.

```
(IMPLIES
 (AND (BVP A) (BVP B) (BOOLP C)
      (EQUAL (LENGTH A) N)
      (EQUAL (LENGTH B) N))
(EQUAL (HEVAL T (CONS (CONS 'V-ADDER N)
                      (CONS 'C (APPEND (GENERATE-NAMES 'A N)
                                       (GENERATE-NAMES 'B N))))
              (PAIRLIST (CONS 'C (APPEND (GENERATE-NAMES 'A N)
                                         (GENERATE-NAMES 'B N)))
                        (CONS C (APPEND A B)))
               (V-ADDER-BOXLIST N))
        (V-ADDER C A B)))
```

The lemmas above have been checked with the Boyer-Moore prover.

The proof of the general lemma has a different flavor than the correctness proofs where the adder is a function in the logic. We began by proving a lemma about the association list returned by the **HEVAL** interpretation of **V-ADDER-BODY**; the structure of the induction is unique to the proof of **V-ADDER-BODY** because of the naming issues. It was also necessary to prove that every output name was assigned exactly once. To ensure that our ripple-carry adder generator function always generates well-formed boxlists, we verified **(BOXLIST-OKP (V-ADDER-BOXLIST N))**.

Although simple, our ripple-carry adder example provides some indication of how circuit generator functions can be written and verified. More sophisticated generator functions can easily be imagined. For instance, we have written a function which generates an adder with a tree-based carry look-ahead scheme. More generally, since the creation of the list constants is program controlled, we are in a position to use any algorithmic means to assist the creation of circuit boxes.

8. An ALU Verification

A circuit box generator function for an *n*-bit ALU is the largest verification we have attempted using a specification in our HDL. This ALU has a specification similar to the FM8502 ALU [Hunt 89a], but the implementation is entirely different in its construction methodology and internal organization. The proof of the ALU circuit box generator function was developed in a hierarchical fashion; subparts of the ALU were first verified and these subparts combined into the final proof.

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: ALU Operation Summary

In the short space of this note we do not attempt to explain the steps of the ALU verification, but we do present an informal specification for the ALU (Table 1) and a lemma stating its correctness. We present this summary of the ALU verification to demonstrate a realistic use of our HDL for specifying a large circuit.

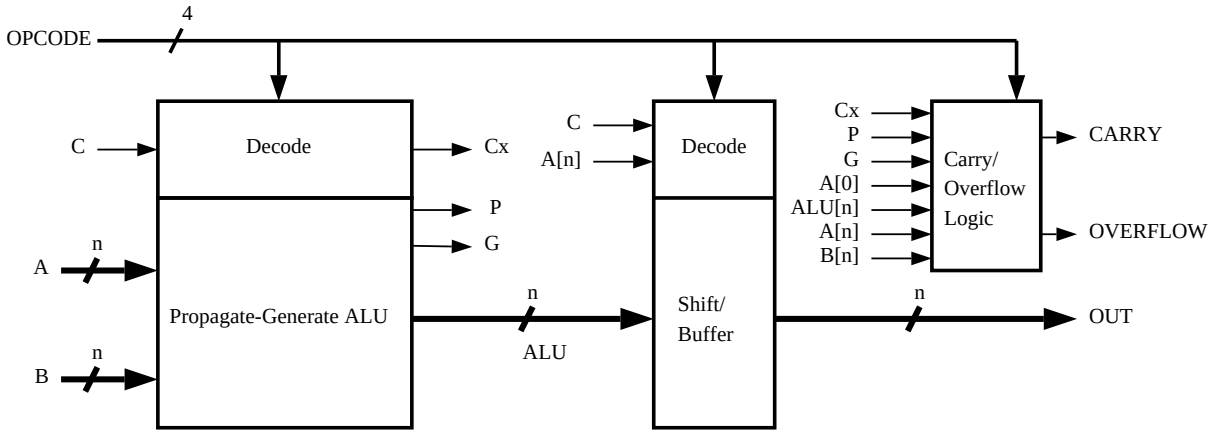

Figure 4: Internal Organization of the ALU

Our ALU is a 16-function unit which produces three outputs: an *n*-bit result, a carry, and an overflow. Our ALU generator function **NEW-ALU-BOXLIST** generates a boxlist which contains the complete circuit description of an *n*-bit ALU. Our specification function **V-ALU** takes two bit-vectors, a carry, and an op-code as arguments. A pictorial representation of the major functional units in the ALU design is shown in Figure 4. The argument **TREE** of **NEW-ALU-BOXLIST** is a set of instructions for several circuit generator functions. **NEW-ALU-BOXLIST** builds circuits recursively depending on the structure of **TREE**; for example, the structure of the propagate-generate logic in the ALU is isomorphic to **TREE**. **TREE** is also used to control the insertion of buffers for common control signals; starting from the bottom, buffers are added at interior nodes whenever the fanout of a control signal may exceed eight loads. The correctness lemma for **NEW-ALU** is given below.

```
(IMPLIES
 (AND (EQUAL BOXLIST (NEW-ALU-BOXLIST TREE))
      (EQUAL (LENGTH A) (TREE-SIZE TREE))
      (EQUAL (LENGTH B) (TREE-SIZE TREE)))
 (EQUAL (HEVAL T (CONS (CONS 'NEW-ALU TREE)
                       (CONS C
                             (APPEND A
                                     (APPEND B
                                             (LIST OP0 OP1
                                                   OP2 OP3)))))
                ALIST BOXLIST)
        (V-ALU (EVAL-NAME C ALIST)
               (COLLECT-EVAL-NAME A ALIST)
               (COLLECT-EVAL-NAME B ALIST)
               (LIST (EVAL-NAME OP0 ALIST)
                     (EVAL-NAME OP1 ALIST)
                     (EVAL-NAME OP2 ALIST)
                     (EVAL-NAME OP3 ALIST)))))
```

Our HDL-based approach has, for the first time, given us the ability to explicitly control circuit fanout, loading, and delay. Given that we use functions to generate our circuits, we are extra sensitive to the quality of circuit we have specified. The gate count, delay, and maximum fanout of our ALU for different word sizes are given in Table 2. The delay of our tree-based, carry look-ahead ALU increases as a constant function of the $\log_2$ of the ALU width.

ALU Characteristics			
Size	**Gate Count**	**Fanout**	**Delay**
1 bit	126	8	12
2 bits	149	8	14
4 bits	196	8	17
8 bits	297	8	22
16 bits	491	8	26
32 bits	880	8	30
64 bits	1665	8	35
128 bits	3227	8	39

Table 2: ALU Result Summary

Processing the proof of the ALU generator functions, including the time necessary to define our primitives and interpreters, takes less than one hour on a Sun 3/280. To generate the circuit boxes for all of the ALU's above takes only a few seconds. For instance, we can generate a complete HDL description of a 32-bit ALU in 1/6 of a second.

9. Conclusions

The formalization of our HDL using list constants provides us with much greater circuit specification precision than we previously enjoyed expressing circuits as functions in the Boyer-Moore logic. We are now able to formally specify well-formed circuits, which was previously impossible. Our interpreters provide logical, delay, and load semantics for circuit boxes. Using this formalization we were able to verify an *n*-bit ALU circuit description by proving that the ALU circuit generating functions produce correct circuits.

The use of constants to represent circuits permits research in verified synthesis and verified tools. Since a circuit specification is just a constant, we can write programs which manipulate these specifications. For example, we could write a logic minimizer and prove the correctness of this program. An algebraic heuristic was employed to control the specification of the fanout for our *n*-bit ALU.

Using formal logic constants to represent circuit boxes provides clear benefits over modeling circuits as formulas in a particular logic: well-formed circuits can be formally specified, circuits can be manipulated with programs in the formalization logic, and verified synthesis and circuit box tools are possible. Circuit generator functions provide a rapid means of producing provably correct circuits. We intend to extend our HDL to include sequential logic, and we plan to explore the use of heuristics in circuit generator functions more thoroughly in the future.

References

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

[Brown 89] Geoffrey M. Brown and Miriam E. Lesser.
From Programs to Transistors: Verifying Hardware Synthesis Tools.
In Miriam Leeser and Geoffrey Brown (editors), *Lecture Notes in Computer Science*. Volume (to appear): *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects.*, pages 128-150. Springer Verlag, 1989.

[Goguen 89] Joesph A. Goguen.
OBJ as a Theorem Prover with Applications to Hardware Verification.
In G. Birtwistle and P.A. Subramanyam (editors), *Current Trends in Hardware Verification and Automatic Theorem Proving*, pages 218-267. Springer-Verlag, New York, 1989.

[Gordon 83] Mike Gordon.
Proving a Computer Correct.
Technical Report 42, University of Cambridge, Computer Laboratory, 1983.

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

[Gordon 86] Mike Gordon, Albert Camilleri, Tom Melham.
Hardware Verification Using Higher-Order Logic.
Technical Report 91, University of Cambridge Computer Laboratory, September, 1986.

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

[Hunt 89a] Warren A. Hunt, Jr. and Bishop C. Brock.
The Verification of a Bit-Slice ALU.
In *Lecture Notes in Computer Science*. Volume (to appear): *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects.*, pages 281-305. Springer Verlag, 1989.

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

[Johnson 89] Steven D. Johnson.
Manipulating Logical Organization with System Factorizations.
In Miriam Leeser and Geoffrey Brown (editors), *Lecture Notes in Computer Science*. Volume (to appear): *Hardware Specification, Verification and Synthesis: Mathematical Aspects.*, pages 259-280. Springer Verlag, 1989.

[Pace 89] Bill Pace and Mark Saaltink.
Formal Verification in m-EVES.
In G. Birtwistle and P.A. Subramanyam (editors), *Current Trends in Hardware Verification and Automatic Theorem Proving*, pages 268-302. Springer-Verlag, New York, 1989.

[Sheeran 84] Mary Sheeran.
μ*FP - An Algebraic VLSI Design Language*.
Technical Report PRG-39, Oxford Programming Research Group, September, 1984.

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

[Viper 89] Avra Cohn.
Correctness Properties of the VIPER Block Model: The Second Level.
In G. Birtwistle and P.A. Subramanyam (editors), *Current Trends in Hardware Verification and Automatic Theorem Proving*, pages 1-91. Springer-Verlag, New York, 1989.

Table of Contents

List of Figures

List of Tables