A Formal Introduction to a Simple HDL

Bishop C. Brock and Warren A. Hunt, Jr.

Technical Report 60 July 31, 1990

Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas 78703-4776

TEL: +1 512 322 9951
FAX: +1 512 322 0656

EMAIL: brock@cli.com, hunt@cli.com

This paper appears in the proceedings of the Formal Methods for VLSI Design Workshop held in Denmark in July 1990. This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151. The views and conclusions contained in this document are those of the author(s) 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. Instead of using logic formulas to represent circuits, we represent circuits as list constants. Using a formal logic, interpreters have been defined which give meanings to circuit constants, and a good-circuit predicate recognizes well-formed circuit descriptions. We can directly verify circuit specifications, but instead we often verify functions which generate circuit constants.

1 Introduction

The formalization of a hierarchical, occurrence-oriented, combinational hardware description language (HDL) has been accomplished using the Boyer-Moore logic. Circuits are represented as Boyer-Moore list constants, and the Boyer-Moore logic [2] is used to define the semantics and syntax of our circuit constants. Instead of verifying each circuit directly, we often prove the correctness of functions which synthesize circuit constants. We employ the Boyer-Moore theorem prover to mechanically manage our database of definitions and to check our proofs.

CAD vendors have long provided tools which operate on circuit descriptions. The typical paradigm is to record design information in a computer data file and write programs to manipulate this data. Programs often translate data from one form to another, e.g., from a gate-graph to a transistor wiring list. For logistical reasons, the formal methods community has not previously attempted to represent circuits as data nor to verify circuits represented as data; circuits have been modeled as terms in a formal logic [10,19,3,7,12,18]. Representing circuits as data involves the definition of interpreters which provide the semantics for the circuit data. To prove the correctness of circuits represented as data requires proofs about the interpretations of the data.

Previously, we used Boyer-Moore logic expressions to represent hardware circuits. Circuits were verified by proving that they satisfied some more abstract specification. However, this approach does not provide a direct migration path to CAD languages, from which an actual physical device can be realized, unless the modeling language is a CAD language itself. Although it may seem that just using an existing CAD language would provide both the modeling capability and an implementation path, commercial CAD languages do not have formal semantics, which means that circuit verification is impossible. Our approach here is to formalize a subset of a conventional CAD language. This approach provides a formal circuit semantics, a formal circuit syntax, and a means of translating circuit descriptions from/to a CAD language.

Here we describe a formalization of a combinational hardware description language and show how we verify functions which generate correct circuit descriptions. Our HDL definition formalizes the notion of circuit delay, fanout, logical values, circuit loading, circuit modules, and circuit module hierarchy; these issues cannot be formally addressed if circuits are modeled with logic formulas. The power of being able to reason about circuits expressed as data cannot be over-emphasized. For instance, we have proved the correctness of a function which produces ALU circuits. We know in advance that any circuit constructed by this ALU-producing function is correct. In addition, modeling circuit specifications with data even admits the possibility of verifying tools (e.g., minimizers, tautology checkers, etc.) which manipulate circuit expressions.

Our presentation begins with an introduction to the notion of hardware verification. We then describe the concept of functions which generate provably

correct circuits, at which point the reader should understand the concepts that the rest of the paper makes more precise.

2 Hardware Verification

We consider hardware verification to mean the use of formal methods for specifying and verifying the operation of digital computing devices. The hardware verification community is attempting to formalize as much of the digital design process as possible. Presently, engineers are given or create specifications that contain a mixture of formal and informal notations from which they are expected to create working devices. The "hardware verification" approach advocates the use of formal logic for both designs and high-level specifications. We introduce this type of approach here.

The use of formal techniques to design hardware is spreading (for example [12,3,4,5,6,18,11,14,17]). Our effort has been ongoing since 1985, and our long-term goal is to provide a means whereby circuits may be rigorously specified and mechanically verified. We conceptualize this notion as providing a mathematical statement, which we call a *formula manual*, that completely specifies the operation of a hardware component. To visualize this notion, imagine a microprocessor user's manual containing a series of formulas that describes the programming model, the timing diagrams, the memory interface, the pin-out, the power requirements, cooling requirements, etc. Further, imagine that available implementations of this microprocessor were verified to meet every specification contained in the formula manual. Then the formula manual would represent a formal specification that would allow hardware engineers to connect this device with complete confidence and would allow software engineers to completely predict the results of programming it.

Formula manuals are a long way off. To address the difficulties of designing, specifying, and constructing real computing equipment, the hardware verification community must continually expand its modeling efforts to explicitly include all hardware attributes which contribute to the correct operation of hardware devices. This expansion will eventually enable us to provide formula manuals for many hardware devices. Formal hardware specification and verification effort has primarily concentrated on the logical correctness of circuit designs. With the formalization of an HDL, we are expanding our formal model to explicitly model (with varying degrees of precision) circuit fanout, loading, and circuit hierarchy. Our desire to explicitly model more hardware characteristics is part of what drove us to consider the specification of an HDL.

Verified hardware design specifications must be convertible to a form that results in physical devices. Formal hardware design specifications are usually described as netlists of transistors or Boolean gates. This level of description is far removed from actual physical implementation; therefore, the conversion of the formal hardware design specifications into physical descriptions is an area

where errors can easily be made. We have designed the HDL presented here to be structured just like some of the HDLs used in commercial CAD systems. It is important that the conversion from verified designs to CAD systems be made at the lowest level possible and be made in a simple and believable way. This is another factor that influenced the design of our HDL.

3 The Boyer-Moore Logic

The HDL we have defined is expressed in terms of Boyer-Moore list constants. We use the Boyer-Moore logic to recognize well-formed HDL expressions and to provide a semantics for our HDL. Here we give a quick introduction to the Boyer-Moore logic and present some examples of its use.

The Boyer-Moore logic [2] is a quantifier-free, first-order predicate calculus with equality. Recursive functions may be defined, provided they terminate. Logic formulas are written in a prefix-style, Lisp-like notation. The basic logic includes several built-in data types: Booleans, natural numbers, lists, literal atoms, and integers. Additional data types can be defined.

An unusual feature of the Boyer-Moore logic is the ability to extend the logic 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 [13] program that provides a user with various commands to extend the logic and to prove theorems. A theorem prover user enters commands through the top-level Common Lisp interpreter. The theorem prover manages the axiom database, user definitions and data types, 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, and it includes a simplifier and rewriter. The theorem prover also contains procedures for automatically performing structural inductions.

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

3.1 Bit-Vectors

We represent a bit-vector as a list of Boolean elements. We formalize this notion with the Boyer-Moore logic by defining the functions `BOOLP` and `BVP`. In this presentation we write definitions with the "`=`" symbol, while theorems are presented without the "`=`" symbol. `BOOLP` tests that `X` is either `T` (true) or `F`

(false).[1] BVP has been defined to recognize a (possibly empty) list of Boolean values. Lists are formed with the pairing function CONS; CAR selects the first element of a pair and CDR the second. For instance, (LIST T F F) is a three-element bit-vector. BVP works recursively; either X is equal to NIL, or (CAR X) is Boolean and (CDR X) is a bit vector as recognized by BVP.

```
(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)))
```

To access a bit in a bit vector we use the function BITN. The Nth bit of LIST is returned if this bit is Boolean; otherwise, F is returned. We will use this formalization of Booleans and bit vectors throughout this paper.

The function APPEND is used to append two list. We can prove that appending two bit vectors together produces a bit vector, and a lemma stating this fact is shown below the definition for APPEND.

```
(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. At this point, a Boyer-Moore theorem prover user would have to have entered the definitions for BOOLP and BVP, and entered the proof command containing the lemma just above.[2]

3.2 Boyer-Moore List Constants and Basic Definitions

Our hardware circuit boxes are represented with Boyer-Moore list constants. Constants are written using the LISP quote notation. The following statements

[1] The semantics for OR, EQUAL, T, and F are described in Boyer and Moore's book, *A Computational Logic Handbook* [2]; see this book for a complete introduction to the Boyer-Moore logic and theorem prover.

[2] The definitions of APPEND and many other simple functions are standard in the Boyer-Moore theorem prover.

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 to provide a structure for our circuit descriptions; components are accessed using combinations of `CAR`s and `CDR`s. `CAR`/`CDR` nests are abbreviated; for example, we write `(CAR (CDR (CDR X)))` as `(CADDR X)`.

Included below are functions used repeatedly throughout the remainder of this paper. These definitions should be skipped upon a first reading, and referred to as needed. As with the definitions above, we consider these definitions to be "obviously correct," that is, we use these functions without proof. `NLISTP` is a predicate which returns true if `X` is not a list. `BOOLFIX` coerces `X` to a Boolean value. `FIRSTN` collects the first `N` bits of a list. `RESTN` collects bits starting at position `N` in `L`. `ASSOC` searches for the key `X` in association list `ALIST` and returns a key-value pair or `F`. `COLLECT-ASSOC` returns a list of values for the keys in `ARGS`. `MEMBER` tests whether `X` is an element of `LIST`. `DISJOINT` is true if no member of `L1` is a member of `L2`. `DUPLICATES?` returns true if no member of `L` occurs twice in `L`. The length of a list is given by `LENGTH`.

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

(BOOLFIX X) = (IF X T F)

(FIRSTN N L) = (IF (LISTP L)
                   (IF (ZEROP N)
                       NIL
                       (CONS (CAR L) (FIRSTN (SUB1 N) (CDR L))))
                   NIL)

(RESTN N L) = (IF (LISTP L)
                  (IF (ZEROP N)
                      L
                      (RESTN (SUB1 N) (CDR L)))
                  L)
```

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

(COLLECT-ASSOC ARGS ALIST) = (IF (NLISTP ARGS)
                                 NIL
                                 (CONS (CDR (ASSOC (CAR ARGS) ALIST))
                                       (COLLECT-ASSOC (CDR ARGS)
                                                      ALIST)))

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

(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))))

(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)))
```

```
(UNION L1 L2) = (IF (LISTP L1)
                    (IF (MEMBER (CAR L1) L2)
                        (UNION (CDR L1) L2)
                        (CONS (CAR L1)
                              (UNION (CDR L1) L2)))
                    L2)

(MAX-MEMBER LIST) = (IF (NLISTP LIST)
                        0
                        (MAX (CAR LIST)
                             (MAX-MEMBER (CDR LIST))))

(MAKE-BALANCED-TREE N)
 =
(IF (ZEROP N)
    0
    (IF (EQUAL N 1)
        0
        (CONS (MAKE-BALANCED-TREE (QUOTIENT N 2))
              (MAKE-BALANCED-TREE (DIFFERENCE N (QUOTIENT N 2))))))

(TREE-SIZE TREE) = (IF (NLISTP TREE)
                       1
                       (PLUS (TREE-SIZE (CAR TREE))
                             (TREE-SIZE (CDR TREE))))

(TFIRSTN LIST TREE) = (FIRSTN (TREE-SIZE (CAR TREE)) LIST)

(TRESTN LIST TREE) = (RESTN (TREE-SIZE (CAR TREE)) LIST)
```

`PAIRLIST` creates an association list. `PROPERP` checks that a list ends with `NIL`. If the members of list `L1` are a subset of the members of `L2`, then `(SUBSET L1 L2)` is true. The function `UNION` produces the set union of `L1` and `L2`. `MAX-MEMBER` returns the maximum number in `LIST`. `MAKE-BALANCED-TREE` constructs a balanced binary tree with `N` leaves. `TREE-SIZE` counts the number of leaves in a binary tree. `TFIRSTN` and `TRESTN` measure the number of leaves in the left-hand part of `TREE` to select elements out of `LIST`.

We have defined a number of small programs which we later use to compute values for our HDL primitives. The definitions below are not a part of the HDL we define later, but are used to help define the logical interpreter for our HDL primitives.

```
(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))
```

4 Introduction to Circuit Generators

The purpose of circuit generator functions is to create parameterized libraries of circuits that can be generated when needed. We call our circuit constructor functions *generators* instead of *synthesizers* because we usually think of synthesis as more fully exploring the design space than do our generator functions. That is not to say that our generator functions are conceptually any different than circuit synthesis functions; just that our generators may be simpler and are proven to be correct.

To give the flavor of our HDL formalization and the construction of verified circuit generator functions, we begin with the definition of an n-bit, ripple-carry adder generator. Later, we will consider the verification of this and several other circuit generator functions.

We represent circuits as a list of circuit boxes (modules). A well-formed circuit box contains four elements: a box name, a list of input names, a list of output names, and a circuit box body. We require the input and output names to be distinct. A circuit box body is just a set of wiring instructions interconnecting circuit boxes. The definition of our HDL includes simple Boolean gate as primitives. These gates are treated as pre-defined circuit boxes. A well-formed circuit box body does not admit wiring loops; i.e., only combinational logic without feedback is permitted. We think of circuit box input and output names as representing wire names.

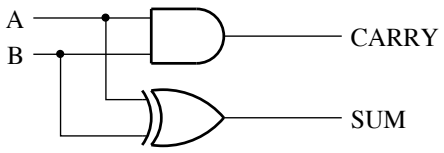

Figure 1: Half-Adder Circuit

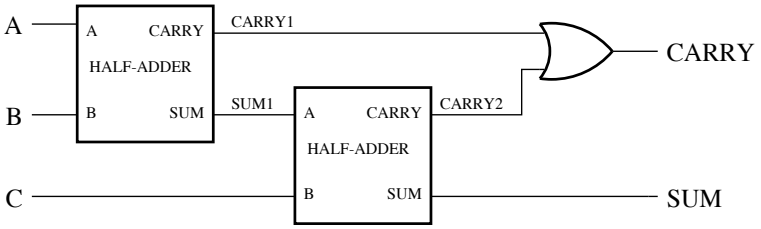

Figure 2: Full-Adder Circuit

Below is a circuit box constant for the half-adder whose schematic diagram is pictured in Figure 1.

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

Circuit box `HALF-ADDER` has two inputs named `A` and `B` and two outputs named `SUM` and `CARRY`. The circuit box body is a list. Each circuit box body occurrence is composed of two elements: a list of outputs and a circuit box reference. Thus the first circuit box body occurrence is `((SUM) (B-XOR A B))`; the output of circuit box reference `(B-XOR A B)` is connected to wire `SUM`.

A schematic for a full-adder is presented in Figure 2. The `FULL-ADDER` circuit box references the `HALF-ADDER` circuit box twice; thus to construct a full-adder circuit requires two half-adders.

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

We introduce the internal wires `SUM1`, `CARRY1`, and `CARRY2` to interconnect the half-adders and the primitive `B-OR` gate.

One way to create an n-bit ripple-carry adder is to connect n full-adders together. For example, Figure 3 is a schematic diagram of a 4-bit ripple-carry

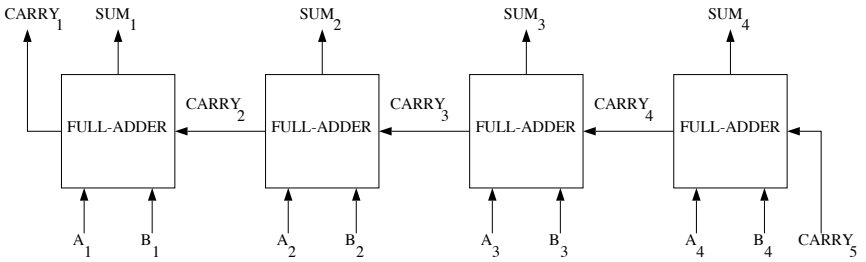

Figure 3: Four-bit, Ripple-carry Adder Circuit

adder, where $\texttt{A}_1$ is the most significant bit and the adder carries from right to left. The circuit box constant for this adder is below.

```
'(RIPPLE-ADDER4
  (CARRY5
   A4 A3 A2 A1
   B4 B3 B2 B1)
  (SUM4 SUM3 SUM2 SUM1 CARRY1)
  (((SUM4 CARRY4) (FULL-ADDER A4 B4 CARRY5))
   ((SUM3 CARRY3) (FULL-ADDER A3 B3 CARRY4))
   ((SUM2 CARRY2) (FULL-ADDER A2 B2 CARRY3))
   ((SUM1 CARRY1) (FULL-ADDER A1 B1 CARRY2))))
```

More generally, we can define a function which creates a circuit box, with an input variable that specifies the adder size. For instance, an n-bit circuit box might be written as follows.

```
'(RIPPLE-ADDERn
  (CARRYn+1
   An ... A1
   Bn ... B1)
  (SUMn ... SUM1 CARRY1)
  (((SUMn CARRYn) (FULL-ADDER An Bn CARRYn+1))
   ...
   ((SUM1 CARRY1) (FULL-ADDER A1 B1 CARRY2))))
```

We construct our ripple-carry adder box generator in two parts: a top-level function which provides the box name, the input names, and the output names; and a function which creates the ripple-carry adder body. The function `(V-ADDER-BODY N)`, just below, creates a list of `N` occurrences where each occurrence is a list of two elements: a list with two output names, $\texttt{SUM}_i$ and $\texttt{CARRY}_i$; and a reference to a full-adder with parameters $\texttt{A}_i$, $\texttt{B}_i$, and $\texttt{CARRY}_{i+1}$. Our Boyer-Moore formulation of `V-ADDER-BODY` does not actually produce terms with subscripts as appears above; we simply abbreviate names of the form `(LIT . `i`)` as $\texttt{LIT}_i$.

```
(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)))))
```

Below is the result of executing `(V-ADDER-BODY 4)`. We use the symbol "`==`" to denote the result of evaluating a function.

```
(V-ADDER-BODY 4)
 ==
'((((SUM . 4) (CARRY . 4))
   (FULL-ADDER (A . 4) (B . 4) (CARRY . 5)))
  (((SUM . 3) (CARRY . 3))
   (FULL-ADDER (A . 3) (B . 3) (CARRY . 4)))
  (((SUM . 2) (CARRY . 2))
   (FULL-ADDER (A . 2) (B . 2) (CARRY . 3)))
  (((SUM . 1) (CARRY . 1))
   (FULL-ADDER (A . 1) (B . 1) (CARRY . 2))))
```

To complete our ripple-adder generator function, we now just need to put the `V-ADDER-BODY` into a circuit box with the appropriate input and output names; the function `V-ADDER*` produces such a circuit box. The function `(GENERATE-NAMES LETTER N)` produces a list of `N` names with `LETTER` as a seed.

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

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

Below is the result of evaluating `(V-ADDER* 4)`.

```
(V-ADDER* 4)
 ==
 '((V-ADDER . 4)                                          ; Name

   ((CARRY . 5)                                           ; Inputs
    (A . 4) (A . 3) (A . 2) (A . 1)
    (B . 4) (B . 3) (B . 2) (B . 1))

   ((SUM . 4) (SUM . 3) (SUM . 2) (SUM . 1)               ; Outputs
    (CARRY . 1))


   ((((SUM . 4) (CARRY . 4))                              ; Body
     (FULL-ADDER (A . 4) (B . 4) (CARRY . 5)))
    (((SUM . 3) (CARRY . 3))
     (FULL-ADDER (A . 3) (B . 3) (CARRY . 4)))
    (((SUM . 2) (CARRY . 2))
     (FULL-ADDER (A . 2) (B . 2) (CARRY . 3)))
    (((SUM . 1) (CARRY . 1))
     (FULL-ADDER (A . 1) (B . 1) (CARRY . 2)))))
```

A complete circuit description is represented as a list of circuit boxes, which we call a *boxlist*. A complete boxlist for an n-bit, ripple-carry adder at the least contains a circuit box generated by `(V-ADDER*` n`)`, along with an instance of `FULL-ADDER` and `HALF-ADDER`.

5 Boxlist Syntax

In this section we present formal specifications of well-formed circuit boxes and boxlists. We consider our simple HDL to be a formal abstraction of a combinational subset of a generic commercial CAD language. We believe that a boxlist that meets our syntactic criteria can be easily and mechanically translated into a commercial CAD language, and used as a reliable basis for a hardware realization of the formal model. We present an example of such a translator in Section 10. We have also defined a number of interpreters for our boxlists including interpreters for circuit values, fanout, and gate count. These interpreters are presented in Section 6. Since Boyer-Moore logic functions are total, these interpreters will compute a result for any boxlist; however, the interpretations are only meaningful for circuit descriptions that meet our syntactic requirements.

The following is an informal summary of the syntax requirements that our boxlist specification makes precise.

- Module names are unique to a boxlist.

```
(BOXLIST-OKP BOXLIST)
 =
(IF (NLISTP BOXLIST)
    (EQUAL BOXLIST NIL)

    (LET ((BOX (CAR BOXLIST))
          (REST (CDR 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 REST))
                  (PROPERP INPUTS)
                  (NOT (DUPLICATES? INPUTS))
                  (PROPERP OUTPUTS)
                  (NOT (DUPLICATES? OUTPUTS))
                  (DISJOINT INPUTS OUTPUTS)
                  (BODY-OKP BODY INPUTS OUTPUTS REST)
                  (BOXLIST-OKP REST))))))
```

Figure 4: Definition of `BOXLIST-OKP`.

- Modules are defined in terms of a small set of combinational primitives, or other modules defined in the boxlist. The boxlist contains a complete hierarchical description of every box.
- Input and output arities are consistent for each primitive and hierarchical reference.
- All nets are driven either by a module input or by exactly one primitive output. There are no busses.
- There is no feedback or potential feedback. In a top-to-bottom scan of the occurrence list for a module, all of the inputs nets for an occurrence must be either module inputs, or appear as outputs of previous occurrences.

The formal specification of a well-formed boxlist is given by `(BOXLIST-OKP BOXLIST)`. The definition of `BOXLIST-OKP` appears as Figure 4. `BOXLIST-OKP` checks

that the boxlist as a whole has the correct structure, that boxes have unique names, and that the inputs and outputs names of each box are unique and disjoint.

The syntax of the occurrence list of each box is specified by `(BODY-OKP BODY SIGNALS OUTPUTS REST)`. The definition of `BODY-OKP` appears as Figure 5. The `BODY` argument of `BODY-OKP` is the occurrence list; `SIGNALS` is initialized to the module input list by `BOXLIST-OKP`, and collects the internal signal names; `OUTPUTS` are the module output signals; and `REST` is the remainder of the boxlist, used to insure that all referenced submodules are defined.

`BODY-OKP`, as well as our interpretation functions defined in the next Section, refer to the specification `(PRIMP FN)`. `PRIMP` defines those names that are considered primitive, and returns a pair `(`*input-arity* `.` *output-arity*`)` for primitives, otherwise `F` for non-primitives.

```
(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))
```

The choice of primitives is completely arbitrary; we have chosen a set of primitives that correspond to the Boolean specification functions displayed in Section 3.1.

6 Hardware Interpreters

To provide a meaning to our HDL circuit constants we have constructed four interpreters: a value interpreter, a delay interpreter, a loading interpreter, and an interpreter which counts the number of primitive references required by a circuit. Each of these interpreters share a similar structure, as each take well-formed circuits as input. Before giving the interpreter definitions, we consider the interpretation of the four-bit adder circuit presented earlier.

The value interpreter produces a logical value for a circuit reference given a boxlist and an association list containing values for signal names. `BOXLIST-OKP` only recognizes circuits which can be evaluated with a single depth first traversal. Each traversal terminates in the evaluation of a primitive. Consider the schematic representation of $\texttt{RIPPLE-ADDER}_4$ presented in Figure 6. To evaluate this circuit, each input must be bound to a value. To evaluate $\texttt{RIPPLE-ADDER}_4$, we

```
(BODY-OKP BODY SIGNALS OUTPUTS REST)
 =
(IF (NLISTP BODY)
    ;;  If the occurrence list is empty, then each output must
    ;;  have been assigned.
    (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))       ;These checks prohibit
          (DISJOINT LHS SIGNALS)        ;busses.
          (LISTP RHS)
          (PROPERP RHS)

          (LET ((MODULE-NAME (CAR RHS))
                (MODULE-ARGS (CDR RHS)))
            (LET ((PRIMP (PRIMP MODULE-NAME)))

              (AND
               (SUBSET MODULE-ARGS SIGNALS) ;Inputs must be driven.
               (IF PRIMP
                   ;;  Arity checks for primitives.
                   (AND (EQUAL (CAR PRIMP) (LENGTH MODULE-ARGS))
                        (EQUAL (CDR PRIMP) (LENGTH LHS)))
                   ;;  Existence and arity checks for submodules.
                   (LET ((SUBMODULE (ASSOC MODULE-NAME REST)))
                     (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 REST)))))))))
```

Figure 5: Definition of `BODY-OKP`.

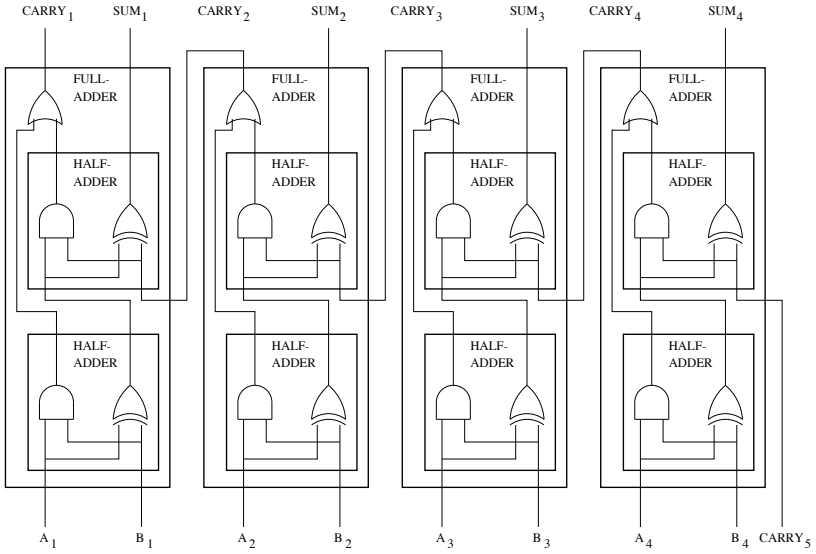

Figure 6: Exploded, Four-bit, Ripple-carry Adder Circuit

must evaluate the four `FULL-ADDER` circuit boxes that comprise $\texttt{RIPPLE-ADDER}_4$. The evaluation begins with the least significant `FULL-ADDER`; the evaluation of `FULL-ADDER` proceeds in a manner similar to $\texttt{RIPPLE-ADDER}_4$. The evaluation continues until left with circuit boxes containing only primitives. The evaluation of primitives is defined by our circuit box evaluator.

6.1 The Logical Value Interpreter

Our logical value interpreter uses association lists to bind values to names. A value for `NAME` in the binding environment `ALIST` is defined by `EVAL-NAME`. `COLLECT-EVAL-NAME` uses `EVAL-NAME` to map a list of names to 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)))
```

We give an example of the use of `COLLECT-EVAL-NAME` below.

```
(COLLECT-EVAL-NAME '(B C A)
                   (LIST (CONS 'A F)
                         (CONS 'B T)
                         (CONS 'C T)
                         (CONS 'D F)))
 ==
(LIST T T F)
```

The specification of evaluating primitive circuit boxes is given by the function `HAPPLY`. `HAPPLY` specifies a value for each function name recognized by `PRIMP`.

```
(HAPPLY FN ACTUALS)
 =
(LET ((A (CAR ACTUALS))
      (B (CADR ACTUALS))
      (C (CADDR ACTUALS))
      (D (CADDDR ACTUALS)))
  (CASE FN
    (B-BUF   (LIST (B-BUF   A)))
    (B-NOT   (LIST (B-NOT   A)))
    (B-NAND  (LIST (B-NAND  A B)))
    (B-NAND3 (LIST (B-NAND3 A B C)))
    (B-NAND4 (LIST (B-NAND4 A B C D)))
    (B-OR    (LIST (B-OR    A B)))
    (B-OR3   (LIST (B-OR3   A B C)))
    (B-OR4   (LIST (B-OR4   A B C D)))
    (B-EQUV  (LIST (B-EQUV  A B)))
    (B-XOR   (LIST (B-XOR   A B)))
    (B-AND   (LIST (B-AND   A B)))
    (B-AND3  (LIST (B-AND3  A B C)))
    (B-AND4  (LIST (B-AND4  A B C D)))
    (B-NOR   (LIST (B-NOR   A B)))
    (B-NOR3  (LIST (B-NOR3  A B C)))
    (B-NOR4  (LIST (B-NOR4  A B C D)))
    (OTHERWISE F))))
```

The evaluation of a circuit box involves alternatively evaluating circuit box references and evaluating circuit box bodies. The Boyer-Moore logic only allows

```
(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))
```

Figure 7: The Logical Value Interpreter, HEVAL.

definitions to refer to previously defined functions; therefore, we are unable to introduce two mutually recursive evaluation functions, but are forced to introduce a single function, HEVAL, containing both a circuit reference evaluator and a circuit box body evaluator. The definition of HEVAL appears as Figure 7. The control flag, FLAG, is used to specify the selection of an evaluator. If FLAG is not F, then FORM is assumed to be a circuit box reference and HEVAL returns a list of values, one value for each output of the reference circuit box. If FLAG is F, the FORM is assumed to be a circuit box body and HEVAL returns an association list that binds values for all outputs.

If FLAG is not F and the occurrence is primitive, then HAPPLY is called. If FLAG is not F and the occurrence is not primitive, then the circuit box body is evaluated by HEVAL with an initial association list that binds the circuit box inputs to the actual parameters. If FLAG is F, HEVAL interprets FORM as a circuit box body;

HEVAL performs the body evaluation recursively by evaluating each circuit box reference in a binding environment that contains the input bindings and output bindings for previously occurring circuit box references. After each reference in a circuit box body has been evaluated, an association list is returned containing bindings for every internal signal along with the original input bindings.

The termination of HEVAL is guaranteed by a decrease in a lexicographic measure of the arguments BOXLIST and FORM with each recursive call of HEVAL. In the call to HEVAL when FLAG is not F, the size of (CDR BOXLIST) is less than the size of BOXLIST. When FLAG is F, the size of FORM is decreasing in both recursive calls of HEVAL: the (CDR FORM) case is obvious, and in the other case HEVAL is called with FORM bound to RHS which is a subcomponent of OCCURRENCE, which itself is a subcomponent of FORM.

To demonstrate a HEVAL interpretation, we consider the evaluation of '(HALF-ADDER A B) in 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))
```

Below is a partial trace of the evaluation of '(HALF-ADDER X Y), 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)) '()))
 ==

(COLLECT-EVAL-NAME '(SUM CARRY)
                   (HEVAL F '(((CARRY) (B-AND A B)))
                          (LIST (CONS 'SUM T)
                                (CONS 'A F) (CONS 'B T)) '()))
 ==

(COLLECT-EVAL-NAME '(SUM CARRY)
                   (HEVAL F '()
                          (LIST (CONS 'CARRY F) (CONS 'SUM T)
                                (CONS 'A F) (CONS 'B T))
                          '()))
 ==
```

```
(COLLECT-EVAL-NAME '(SUM CARRY)
                   (LIST (CONS 'CARRY F) (CONS 'SUM T)
                         (CONS 'A F) (CONS 'B T)))
 ==
(LIST T F)
```

6.2 Other Interpreters

We have written several other interpreters: a delay interpreter which computes the delay for every output of any circuit box reference; a load interpreter which computes the input loading, output loading, and maximum internal loading of a circuit box reference; and an interpreter that counts the total number of primitives that must be evaluated for any circuit box reference.

GEVAL counts the number of primitives evaluated for any circuit box reference. For example, the ripple-carry adder circuit in Figure 6 contains 20 primitives gates. The structure of GEVAL is similar to HEVAL's structure. GEVAL's primitive apply function GAPPLY returns 1; however, it is possible to generalize GAPPLY to return different values for different primitives.

When given a circuit box reference, GEVAL calls GAPPLY if FORM is primitive; otherwise, GEVAL computes the number of primitives in the body of the current circuit box reference. Note, when the FLAG is not F, FORM is a circuit box reference, otherwise FORM is a circuit box body.

```
(GAPPLY FN) = 1

(GEVAL FLAG FORM BOXLIST)
 =
(IF FLAG
    (LET ((FN   (CAR FORM))
          (ARGS (CDR FORM)))
      (IF (PRIMP FN)
          (GAPPLY FN)
          (LET ((BOX (ASSOC FN BOXLIST)))
            (IF BOX
                (LET ((BODY (CADDDR BOX)))
                  (GEVAL F BODY (CDR BOXLIST)))
                F))))

    (IF (LISTP FORM)
        (LET ((OCCURRENCE (CAR FORM)))
          (LET ((RHS (CADR OCCURRENCE)))
            (PLUS (GEVAL T RHS BOXLIST)
                  (GEVAL F (CDR FORM) BOXLIST))))
        0))
```

DEVAL computes the maximum delay for every output from every relevant input. Unique initial delays may be provided for every input. In DAPPLY we have chosen the delay for each primitive to be one more than the maximum delay of its inputs. We could define DAPPLY to reflect delays for a particular implementation technology.

```
(DAPPLY FN ACTUALS)
 =
(LIST (ADD1 (MAX-MEMBER ACTUALS)))
```

DEVAL operates in a manner just like HEVAL, but instead of computing the value for every output, it computes the delay for every output. The initial ALIST contains the initial delays for every input.

```
(DEVAL FLAG FORM ALIST BOXLIST)
 =
(IF FLAG
    (LET ((FN   (CAR FORM))
          (ARGS (CDR FORM)))
      (LET ((DELAYS (COLLECT-ASSOC ARGS ALIST)))
        (IF (PRIMP FN)
            (DAPPLY FN DELAYS)
          (LET ((BOX (ASSOC FN BOXLIST)))
            (IF BOX
                (LET ((INPUTS  (CADR BOX))
                      (OUTPUTS (CADDR BOX))
                      (BODY    (CADDDR BOX)))
                  (COLLECT-ASSOC
                   OUTPUTS
                   (DEVAL F BODY (PAIRLIST INPUTS DELAYS)
                          (CDR BOXLIST))))
                F)))))
  (IF (LISTP FORM)
      (LET ((OCCURRENCE (CAR FORM)))
        (LET ((LHS (CAR OCCURRENCE))
              (RHS (CADR OCCURRENCE)))
          (DEVAL F
                 (CDR FORM)
                 (APPEND (PAIRLIST LHS
                                   (DEVAL T RHS ALIST BOXLIST))
                         ALIST)
                 BOXLIST)))
      ALIST))
```

The loading interpreter, LEVAL, returns three items: the input loadings, the output loadings, and the maximum internal loading. Each primitive input has a loading of one. Primitive output loadings are zero; however, the output loadings of a circuit box are not always zero, as box outputs may be wired to other

internal inputs. The loadings for the primitive circuit boxes are specified by LAPPLY. The LEVAL function concerns itself with the structure of a circuit box reference, not about values it may compute. Loadings are returned as a list with the input loading first, the output loading second, and the maximum internal loading last.

```
(LAPPLY FN)
 =
(LIST (MAKE-LIST (CAR (PRIMP FN)) 1) (LIST 0) 1))
```

ALIST is used in LEVAL to supply initial loadings for every input.

```
(LEVAL FLAG FORM ALIST BOXLIST)
 =
(IF FLAG
    (LET ((FN   (CAR FORM))
          (ARGS (CDR FORM)))
      (IF (PRIMP FN)
          (LAPPLY FN)
          (LET ((BOX (ASSOC FN BOXLIST)))
            (IF BOX
                (LET ((INPUTS  (CADR BOX))
                      (OUTPUTS (CADDR BOX))
                      (BODY    (CADDDR BOX)))
                  (LET ((LEVAL (LEVAL F BODY (PAIRLIST INPUTS 0)
                                             (CDR BOXLIST))))
                    (LET ((ALIST (CAR LEVAL))
                          (NEW-MAX-FAN (CDR LEVAL)))
                      (LIST (COLLECT-ASSOC INPUTS ALIST)
                            (COLLECT-ASSOC OUTPUTS ALIST)
                            NEW-MAX-FAN))))
                F))))
    (IF (LISTP FORM)
        (LET ((OCCURRENCE (CAR FORM)))
          (LET ((LHS (CAR OCCURRENCE))
                (RHS (CADR OCCURRENCE)))
            (LET ((TRIPLE (LEVAL T RHS NIL BOXLIST)))
              (LET ((I-COSTS (CAR TRIPLE))
                    (O-COSTS (CADR TRIPLE))
                    (TERM-MAX-COST (CADDR TRIPLE)))
                (LET ((NEW-ALIST
                       (APPEND (PAIRLIST LHS O-COSTS)
                               (ADD-TO-ALIST (CDR RHS)
                                             I-COSTS ALIST))))
                  (LET ((RETURN-ALIST
                         (CAR (LEVAL F (CDR FORM) NEW-ALIST
                                      BOXLIST)))
                        (RETURN-MAX-COST
                         (CDR (LEVAL F (CDR FORM) NEW-ALIST
                                      BOXLIST))))
                    (CONS RETURN-ALIST
                          (MAX TERM-MAX-COST RETURN-MAX-COST)))))))))
        (CONS ALIST (MAX-ALIST-VAL ALIST))))
```

It is easy to imagine the definition of other interpreters. For instance, a

critical path interpreter could be written, delays could computed with fanout information taken into account, or power requirements could be estimated.

7 A Simple Circuit Generator

A consequence of using constants to specify hardware circuits is an ability to verify functions which generate circuit descriptions. To demonstrate verifying circuit generators, we consider the proof of the ripple-carry adder generator presented earlier. The ripple-carry adder example is often used in descriptions of formal hardware verification methodologies; thus the interested reader can directly compare this treatment with other approaches [15,9,19,16]. We are not aware of other work specifically attempting to verify circuit generator functions.

It is important to clarify exactly what we intend to verify. We can only prove the correctness of the interpretation of a circuit box reference with respect to a boxlist that completely defines the hierarchical structure of the circuit. The verification of circuit generators is three part: we specify a circuit box generator, we specify a predicate which recognizes a boxlist that contains instances of the circuit box generator, and finally we prove the correctness of a reference to the circuit box with respect to a specification function, assuming our predicate holds for the boxlist. The structure of these predicates mimics the hierarchy of the design, as do the structure of the proofs. We present the verification of our ripple-carry adder generator in a bottom-up manner.

A lemma stating the correctness of a reference to an exclusive OR primitive is shown below. In the case of primitives, like exclusive OR, we need not construct circuit generators, and the predicates which recognize primitives in a boxlist are always true.

```
(B-XOR& BOXLIST) = T

(IMPLIES (B-XOR& BOXLIST)
         (EQUAL (HEVAL T (LIST 'B-XOR X Y) ALIST BOXLIST)
                (LET ((X (EVAL-NAME X ALIST))
                      (Y (EVAL-NAME Y ALIST)))
                  (LIST (B-XOR X Y)))))
```

This lemma indicates that the evaluation with `HEVAL` of any reference of the form `(LIST 'B-XOR X Y)` is precisely the same as applying the specification function `B-XOR` to the values of `X` and `Y` in `ALIST`. We now take as given that the boxlist recognizers for our primitives are true, and that we have proven the correctness of the evaluation of each primitive with respect to their specification functions.

The verification of a circuit box generator for a half-adder proceeds in three parts: we define the circuit box generator `HALF-ADDER*`, we define a predicate, `HALF-ADDER&`, that recognizes boxlists containing an instance of our half-adder circuit box, and finally we prove the correctness of a reference to a half-adder.

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

(HALF-ADDER& BOXLIST)
 =
(AND (EQUAL (ASSOC 'HALF-ADDER BOXLIST) (HALF-ADDER*))
     (B-XOR& (CDR BOXLIST))
     (B-AND& (CDR BOXLIST)))

(IMPLIES (HALF-ADDER& BOXLIST)
         (EQUAL (HEVAL T (LIST 'HALF-ADDER A B) ALIST BOXLIST)
                (LET ((A (EVAL-NAME A ALIST))
                      (B (EVAL-NAME B ALIST)))
                  (LIST (B-XOR A B)
                        (B-AND A B)))))
```

We verify the correctness of our full-adder in the same manner as we verified our half-adder. We first introduce the specification functions `FULL-ADDER-SUM` and `FULL-ADDER-CARRY` for our full-adder.

```
(FULL-ADDER-SUM A B C)
 =
(IF A
    (IF B (IF C T F) (IF C F T))
    (IF B (IF C F T) (IF C T F)))

(FULL-ADDER-CARRY A B C)
 =
(OR (AND A (OR B C)) (AND B C))
```

Shown below is our full-adder box generator, `FULL-ADDER*`, a predicate which recognizes boxlists that contain an instance of `FULL-ADDER*`, and a lemma demonstrating the correctness of evaluating a reference to a full-adder.

```
(FULL-ADDER*)
 =
'(FULL-ADDER (A B C) (SUM CARRY)
             (((SUM1 CARRY1) (HALF-ADDER A B))
              ((SUM CARRY2) (HALF-ADDER SUM1 C))
              ((CARRY) (B-OR CARRY1 CARRY2)))))
(FULL-ADDER& BOXLIST)
 =
(AND (EQUAL (ASSOC 'FULL-ADDER BOXLIST) (FULL-ADDER*))
     (HALF-ADDER& (CDR BOXLIST))
     (B-OR& (CDR BOXLIST)))

(IMPLIES (FULL-ADDER& BOXLIST)
         (EQUAL (HEVAL T (LIST 'FULL-ADDER A B C) ALIST BOXLIST)
                (LET ((A (EVAL-NAME A ALIST))
                      (B (EVAL-NAME B ALIST))
                      (C (EVAL-NAME C ALIST)))
                  (LIST (FULL-ADDER-SUM A B C)
                        (FULL-ADDER-CARRY A B C)))))
```

We are now ready to verify the correctness of the ripple-carry adder generator with respect to our Boolean addition specification `V-ADDER`.

```
(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))))
```

`V-ADDER` specifies Boolean addition of bit-vectors `A` and `B`, with an initial input carry `C`.

Our ripple-carry adder generator, `V-ADDER*`, generates a circuit box whose body contains `N` references to full-adders; these references are constructed by the helper function `V-ADDER-BODY`.

```
(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
```

We verify the correctness of our helper function V-ADDER-BODY before attempting to verify V-ADDER*. The lemma below shows that the association list returned by HEVAL from the evaluation of the body of the adder binds the output names correctly.

```
(IMPLIES
 (AND (FULL-ADDER& BOXLIST)
      (EQUAL C (EVAL-NAME (CONS 'CARRY (ADD1 N)) ALIST))
      (EQUAL A (COLLECT-EVAL-NAME (GENERATE-NAMES 'A N) ALIST))
      (EQUAL B (COLLECT-EVAL-NAME (GENERATE-NAMES 'B N) ALIST)))
 (EQUAL (COLLECT-EVAL-NAME (APPEND (GENERATE-NAMES 'SUM N)
                                   (LIST (CONS 'CARRY 1)))
                           (HEVAL F
                                  (V-ADDER-BODY N)
                                  ALIST
                                  BOXLIST))
        (V-ADDER C A B)))
```

We prove that the HEVAL interpretation of a reference to a V-ADDER circuit box is equal to Boolean addition only if the BOXLIST contains an instance of V-ADDER* (the adder generator), and instances of the HALF-ADDER and FULL-ADDER circuits. With the interpretation of the adder body in hand, it is a simple matter to complete the proof of adder circuit references.

```
(V-ADDER& BOXLIST N)
 =
(AND (EQUAL (ASSOC (CONS 'V-ADDER N) BOXLIST) (V-ADDER* N))
     (FULL-ADDER& (CDR BOXLIST)))
```

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

We have not demonstrated a well-formed boxlist that satisfies `V-ADDER&`. `V-ADDER-BOXLIST` generates a boxlist that is both well-formed and satisfies `V-ADDER&`.

```
(V-ADDER-BOXLIST N) = (LIST (V-ADDER* N)
                            (FULL-ADDER*)
                            (HALF-ADDER*))
```

`V-ADDER-BOXLIST` generates a boxlist with three circuit boxes: an n-bit ripple-carry adder, a full-adder, and a half-adder. It is trivial to prove that `V-ADDER-BOXLIST` satisfies `V-ADDER&`. It is also possible to prove that `V-ADDER-BOXLIST` generates a well-formed boxlist; however, we do not prove either property. In general, we never prove that our parameterized circuit generators are well-formed or that they satisfy any particular circuit boxlist predicate. Instead, we prove that instances of boxlists created by parameterized boxlist generators are well-formed by executing `BOXLIST-OKP`. Likewise, we show that instances of circuit boxlists satisfy their circuit boxlist predicates by executing these predicates on boxlists of interest. For example, when we generate a boxlist with `V-ADDER-BOXLIST` we check that its result is complete and well-formed by executing the functions `V-ADDER&` and `BOXLIST-OKP`.

We can extend the lemma above by observing that `V-ADDER` can be used to add natural numbers when they are represented as bit-vectors. We define the abstraction function `V-TO-NAT` that converts a bit-vector into a natural number.

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

We are then able to prove the lemma below, which shows that `V-ADDER` specifies addition of bit-vector representations of natural numbers.

```
(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))))
```

With this lemma, we are able to finally prove the lemma below.

```
(IMPLIES (AND (V-ADDER& BOXLIST N)
              (EQUAL (LENGTH A) N)
              (EQUAL (LENGTH B) N))
         (EQUAL (V-TO-NAT (HEVAL T (CONS (CONS 'V-ADDER N)
                                         (CONS C (APPEND A B)))
                                 ALIST BOXLIST))
                (LET ((C (EVAL-NAME C ALIST))
                      (A (COLLECT-EVAL-NAME A ALIST))
                      (B (COLLECT-EVAL-NAME B ALIST)))
                  (PLUS (IF C 1 0)
                        (V-TO-NAT A)
                        (V-TO-NAT B)))))
```

The presentation above sets the theme for the next two sections. In the next section we employ heuristics to generate an adder with a minimum cost. Afterward, we outline an ALU generator.

8 A Heuristically Guided Adder Generator

The use of programs to generate provably correct hardware circuits provides great freedom in selecting designs through the use of heuristic guidance. Here we outline an adder circuit generator that selects a ripple-carry or propagate-generate adder based on a comparison of their costs.

The function `COST` computes the cost of a circuit reference by adding the maximum circuit delay to the result of dividing the number of primitive gates required to build the circuit by three. To aid in the computation of the cost we use `DEVAL` to calculate adder delays and `GEVAL` to compute the number of required gates.

```
(COST FORM BOXLIST)
 =
(PLUS (QUOTIENT (GEVAL T FORM BOXLIST) 3)
      (MAX-MEMBER (DEVAL T FORM (PAIRLIST (CDR FORM) 0) BOXLIST)))
```

We use the circuit box generated by `ADDER*` to provide an indirect reference to either a ripple-carry adder or a propagate-generate adder. The particular reference returned by `ADDER*` is picked by a comparison of the cost for an n-bit

ripple-carry adder with the cost for an n-bit propagate-generate adder. To make this comparison, ADDER* actually creates instances of both the ripple-carry adder and the propagate-generate adder, measures their costs, and returns a reference to the adder with the least cost. Propagate-generate adders are characterized by their tree-based look-ahead logic. To specify a propagate-generate adder of n-bits, we construct a balanced binary tree with n leaves; the structure of this tree specifies the organization of the look-ahead logic.

The type of adder for which an indirect reference is created is specified in the last three lines of the definition of ADDER*. The remainder of ADDER* definition is used to make the comparison between the two adder types.

```
(ADDER* N)
 =
(LET ((TREE (MAKE-BALANCED-TREE N)))
  (LET ((RIPPLE-NAME (CONS 'V-ADDER N))
        (PG-NAME     (CONS 'TV-ADDER TREE))
        (ARGS        (CONS (CONS 'CARRY (ADD1 N))
                           (APPEND (GENERATE-NAMES 'A N)
                                   (GENERATE-NAMES 'B N))))
        (OUTPUTS     (GENERATE-NAMES 'OUT (ADD1 N))))
    (LET ((RIPPLE-BOXLIST (V-ADDER-BOXLIST N))
          (PG-BOXLIST     (TV-ADDER-BOXLIST TREE))
          (RIPPLE-FORM    (CONS RIPPLE-NAME ARGS))
          (PG-FORM        (CONS PG-NAME ARGS)))
      (LET ((RIPPLE-COST (COST RIPPLE-FORM RIPPLE-BOXLIST))
            (PG-COST     (COST PG-FORM PG-BOXLIST)))
        (LIST (CONS 'ADDER N)                              ; Name
              ARGS                                         ; Inputs
              OUTPUTS                                      ; Outputs
              (IF (LESSP RIPPLE-COST PG-COST)              ; Body
                  (LIST (LIST OUTPUTS RIPPLE-FORM))
                  (LIST (LIST OUTPUTS PG-FORM))))))))
```

For example, an instance of ADDER* with N equals four is shown below.

```
(ADDER* 4)
 ==
'((ADDER . 4)
   ((CARRY . 5)
    (A . 4) (A . 3) (A . 2) (A . 1)
    (B . 4) (B . 3) (B . 2) (B . 1))
   ((OUT . 5) (OUT . 4) (OUT . 3) (OUT . 2) (OUT . 1))
   ((((OUT . 5) (OUT . 4) (OUT . 3) (OUT . 2) (OUT . 1))
     ((V-ADDER . 4)
      (CARRY . 5)
      (A . 4) (A . 3) (A . 2) (A . 1)
      (B . 4) (B . 3) (B . 2) (B . 1)))))
```

We prove a lemma that demonstrates the correctness of our adder generator given any boxlist that satisfies `ADDER&`. `TV-ADDER&` is a predicate that recognizes boxlists with instances of propagate-generate adders.

```
(ADDER& BOXLIST N)
 =
(AND (EQUAL (ASSOC (CONS 'ADDER N) BOXLIST) (ADDER* N))
     (LET ((TREE (MAKE-BALANCED-TREE N)))
       (AND (V-ADDER& (CDR BOXLIST) N)
            (TV-ADDER& (CDR BOXLIST) TREE))))

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

This lemma has the same form as the lemma we previously proved for our ripple carry adder. For simplicity, we impose the condition in `ADDER&` that `BOXLIST` contains both ripple-carry and propagate-generate adder definitions.

We have computed the costs for different sized ripple-carry and propagate-generate adders and provide a comparison of these costs in Table 1. The maximum delay and number of gates for a ripple-carry adder increase linearly with the size of the adder. The propagate-generate maximum delay increases logarithmically with the size of the adder, while the size increases at a roughly linear rate. Notice that an adder larger than 26 bits must be generated before the propagate-generate adder becomes cheaper than the ripple-carry adder.

The point to this section is to demonstrate how heuristics can be used to control the generation of provably correct circuits. Our adder generator selects an "appropriate" adder based on our cost function. Note that any other scheme could have been substituted to select which adder is specified.

9 An ALU Generator

We have verified a circuit box generator function for an n-bit ALU [1]. There are two parts to the ALU verification: the proof that the top-level Boolean ALU specification implements mathematical functions and the proof that the HDL circuit generated implements the top-level Boolean ALU specification. Here we present the top-level Boolean ALU specification, its statement of correctness,

Adder Size	Ripple Gate Count	Ripple Max Delay	Ripple Cost	Prop-Gen Gate Count	Prop-Gen Max Delay	Prop-Gen Cost
1 bit	5	3	4	6	3	5
2 bits	10	5	8	15	5	10
4 bits	20	9	15	33	8	19
8 bits	40	17	30	69	12	35
16 bits	80	33	59	132	15	59
25 bits	125	51	92	222	19	93
26 bits	130	53	96	231	19	96
27 bits	135	55	100	240	19	99
32 bits	160	65	118	285	20	115
64 bits	320	129	235	573	24	215
128 bits	640	257	470	1149	28	411

Table 1: Ripple-Carry, Generate-Propagate Adder Cost Comparison

and sketch a part of the internal implementation of the ALU.[3]

Our Boolean ALU specification, `V-ALU`, plays the same role that `V-ADDER` played in the verification of our ripple-carry adder. `V-ALU` is the specification that our ALU circuit generator is verified to implement. An informal summary of `V-ALU` is presented in Table 2.

`V-ALU` is the Boolean specification for our ALU. It requires four inputs: `C`, a Boolean carry input; `A` and `B`, bit-vector inputs; and a four-bit op-code, `OP`. `V-ALU` returns a bit-vector that is two bits longer than bit-vector `A`: the first bit is the carry output, the second bit is the overflow output, and the remainder is the result bit-vector.

[3]We have proved that the top-level Boolean ALU specification implements a number of Boolean, natural number, and integer operations. Each ALU operation has been verified to correctly implement one or more functions. For instance, if a logical OR operation is selected, then we have proved that our Boolean ALU specification computes the logical OR; however, if an arithmetic shift right is selected we have proved not only the logical properties of this operation, but we have also proved that this operation implements a division by two for a bit-vector representing a natural number or an integer. For addition operations we have proved the correctness of the Boolean ALU specification with respect to natural number and integer addition; likewise for subtraction. Here, we give the ALU specification as a Boolean function, as this is what the actual ALU hardware should compute.

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	$a \gg 1$	Rotate right, shifted through carry
1001	$a \gg 1$	Arithmetic shift right, top bit duplicated
1010	$a \gg 1$	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 2: Informal ALU Operation Summary

```
(V-ALU C A B OP)
 =
(COND ((EQUAL OP (BC 'B0000)) (CVBV F F (V-BUF A)))
      ((EQUAL OP (BC 'B0001)) (CVBV-INC A))
      ((EQUAL OP (BC 'B0010)) (CVBV-V-ADDER C A B))
      ((EQUAL OP (BC 'B0011)) (CVBV-V-ADDER F A B))
      ((EQUAL OP (BC 'B0100)) (CVBV-NEG A))
      ((EQUAL OP (BC 'B0101)) (CVBV-DEC A))
      ((EQUAL OP (BC 'B0110)) (CVBV-V-SUBTRACTER C A B))
      ((EQUAL OP (BC 'B0111)) (CVBV-V-SUBTRACTER F A B))
      ((EQUAL OP (BC 'B1000)) (CVBV-V-ROR C A))
      ((EQUAL OP (BC 'B1001)) (CVBV (BITN 0 A) F (V-ASR A)))
      ((EQUAL OP (BC 'B1010)) (CVBV (BITN 0 A) F (V-LSR A)))
      ((EQUAL OP (BC 'B1011)) (CVBV F F (V-XOR A B)))
      ((EQUAL OP (BC 'B1100)) (CVBV F F (V-OR  A B)))
      ((EQUAL OP (BC 'B1101)) (CVBV F F (V-AND A B)))
      ((EQUAL OP (BC 'B1110)) (CVBV F F (V-NOT A)))
      (T                      (CVBV F F (V-BUF A))))
```

Of the many different possible implementations for the `V-ALU` specification, we have verified an ALU implementation which includes a propagate-generate look-ahead scheme for additions and subtractions. Previously, we have verified implementations of `V-ALU` with a ripple-carry approach [19] and a bit-slice

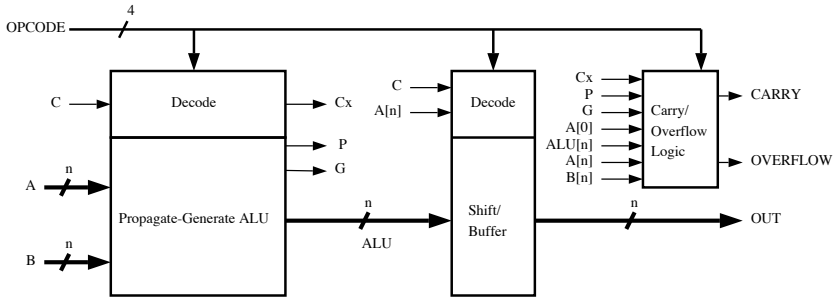

Figure 8: Internal Organization of the ALU

approach [20]; however, these previous approaches did not specify their implementations with an HDL but required a program to extract a netlist from Boyer-Moore definitions which represented the design specification. Our ALU implementation is produced by a generator, like `V-ADDER*`, but it generates quite a large number of boxes with fairly complicated interconnections.

A block diagram of the internal structure of our ALU implementation is shown in Figure 8. The ALU is divided into three main parts: a propagate-generate ALU, which performs logical and arithmetic operations; a shift/buffer unit, which performs the right shift operations; and a carry/overflow unit which computes carry and overflow.

Our boxlist recognizer for our ALU is `NEW-ALU&`, which is presented below. We do not present the individual predicates that `NEW-ALU&` references.

```
(NEW-ALU& BOXLIST TREE)
 =
(AND (EQUAL (ASSOC (CONS 'NEW-ALU TREE) BOXLIST)
            (NEW-ALU* TREE))
     (B-BUF&            (CDR BOXLIST))
     (MPG&              (CDR BOXLIST))
     (TV-SHIFT-OR-BUF& (CDR BOXLIST) TREE)
     (CARRY-IN-HELP&    (CDR BOXLIST))
     (TV-ALU-HELP&      (CDR BOXLIST) TREE)
     (T-CARRY&          (CDR BOXLIST))
     (CARRY-OUT-HELP&   (CDR BOXLIST))
     (OVERFLOW-HELP&    (CDR BOXLIST)))
```

The lemma stating the correctness of our ALU is below.

```
(IMPLIES
 (AND (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)))))
```

That is, when the predicate NEW-ALU& is satisfied, then the HEVAL of $\texttt{NEW-ALU}_{TREE}$ satisfies our specification function V-ALU.

Instead of presenting the entire specification and implementation of our ALU generator, we have chosen to present one particular part of the ALU. The most interesting part of our ALU generator is the propagate-generate arithmetic/logical unit (PG-ALU). The PG-ALU performs all of the ALU operations expect for the shift operations and the computation of the carry and overflow outputs. The PG-ALU circuit has 4 inputs; bit-vector data inputs A and B, a carry input C, and an 8-bit control vector MPG.

The PG-ALU circuit generator requires a single parameter, TREE. The number of leaves in the tree defines the size of the data input vectors, and the structure of the tree determines the configuration of the carry look-ahead logic. Each leaf of the tree is represented in the final circuit by an instance of an ALU function cell, T-CELL. Each internal node of the tree is represented in the final circuit by two PG-ALU modules connected together with carry look-ahead logic. The definition of the PG-ALU generator TV-ALU-HELP* appears as Figure 10. The schematic diagram in Figure 9 shows the interconnections generated by each internal tree node. The definition of the carry look-ahead cell, T-CARRY, is below.

```
(T-CARRY*)
 =
'(T-CARRY (C P G) (COUT)
          (((T0)   (B-AND C P))
           ((COUT) (B-OR G T0))))
```

Notice that TV-ALU-HELP-BODY buffers the control lines only if a control line may drive more than 8 primitives. Also notice that TV-ALU-HELP-BODY is not a

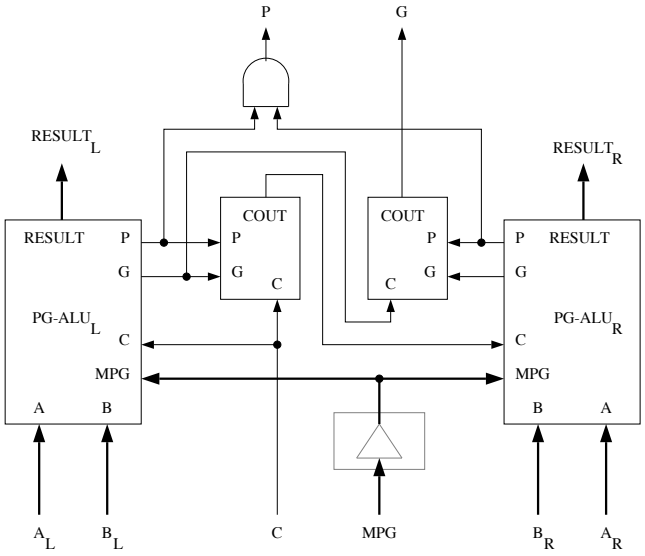

Figure 9: PG-ALU Circuit Organization

recursive function like `V-ADDER-BODY`, but creates a circuit box that references submodules created by other calls to `TV-ALU-HELP*` on subtrees of `TREE`. In other words, the entire boxlist has a recursive structure. This condition is stated formally by the predicate `TV-ALU-HELP&`.

```
(TV-ALU-HELP& BOXLIST TREE)
 =
(IF (NLISTP TREE)
    (AND (EQUAL (ASSOC (CONS 'TV-ALU-HELP TREE) BOXLIST)
                (TV-ALU-HELP* TREE))
         (T-CELL& (CDR BOXLIST)))
    (AND (EQUAL (ASSOC (CONS 'TV-ALU-HELP TREE) BOXLIST)
                (TV-ALU-HELP* TREE))
         (T-CARRY& (CDR BOXLIST))
         (B-AND& (CDR BOXLIST))
         (V-BUF& (CDR BOXLIST) 8)
         (TV-ALU-HELP& (CDR BOXLIST) (CAR TREE))
         (TV-ALU-HELP& (CDR BOXLIST) (CDR TREE))))
```

Using an HDL-based approach provides an ability to explicitly control circuit fanout, loading, and delay. Using the functions `GEVAL`, `DEVAL` and `LEVAL` we have computed the gate count, delay, and maximum fanout of our ALU for different word sizes; these are given in Table 3. The delay of our ALU with its propagate-generate look-ahead logic grows with the $\log_2$ of the size of the ALU when generated with balanced binary trees.

```
(TV-ALU-HELP* TREE)
 =
(LET ((A-NAMES (GENERATE-NAMES 'A (TREE-SIZE TREE)))
      (B-NAMES (GENERATE-NAMES 'B (TREE-SIZE TREE)))
      (OUT-NAMES (GENERATE-NAMES 'OUT (TREE-SIZE TREE)))
      (MPGNAMES (GENERATE-NAMES 'MPG 8)))
  (LIST (CONS 'TV-ALU-HELP TREE)        ;;  Name
        (CONS 'C (APPEND A-NAMES        ;;  Inputs
                         (APPEND B-NAMES MPGNAMES)))
        (CONS 'P (CONS 'G OUT-NAMES))   ;;  Outputs
        (TV-ALU-HELP-BODY TREE)))       ;;  Body

(TV-ALU-HELP-BODY (TREE)
 =
(LET ((A-NAMES (GENERATE-NAMES 'A (TREE-SIZE TREE)))
      (B-NAMES (GENERATE-NAMES 'B (TREE-SIZE TREE)))
      (OUT-NAMES (GENERATE-NAMES 'OUT (TREE-SIZE TREE)))
      (MPGNAMES (GENERATE-NAMES 'MPG 8))
      (MPGNAMES* (GENERATE-NAMES 'MPG* 8)))
 (LET ((LEFT-A-NAMES (TFIRSTN A-NAMES TREE))
       (RIGHT-A-NAMES (TRESTN A-NAMES TREE))
       (LEFT-B-NAMES (TFIRSTN B-NAMES TREE))
       (RIGHT-B-NAMES (TRESTN B-NAMES TREE))
       (LEFT-OUT-NAMES (TFIRSTN OUT-NAMES TREE))
       (RIGHT-OUT-NAMES (TRESTN OUT-NAMES TREE)))
  (LET ((BUFFER? (EQUAL (REMAINDER (SUB1 (TREE-HEIGHT TREE)) 3) 0)))
   (LET ((MPGNAMES? (IF BUFFER? MPGNAMES* MPGNAMES)))
     (IF (NLISTP TREE)
         (LIST
          (LIST '(P G (OUT . 1))
                (CONS 'T-CELL (CONS 'C (CONS '(A . 1)
                                             (CONS '(B . 1)
                                                   MPGNAMES))))))
       ;;  Buffer MPG if more than 8 loads.
       (APPEND
        (IF BUFFER? (LIST
                     (LIST MPGNAMES* (CONS (CONS 'V-BUF 8) MPGNAMES)))
          NIL)
        (LIST
         ;;  The LHS alu
         (LIST (CONS 'PL (CONS 'GL LEFT-OUT-NAMES))
               (CONS (CONS 'TV-ALU-HELP (CAR TREE))
                     (CONS 'C (APPEND LEFT-A-NAMES
                                      (APPEND LEFT-B-NAMES
                                              MPGNAMES?)))))
         ;;  The LHS carry
         '((CL) (T-CARRY C PL GL))
         ;;  The RHS alu
         (LIST (CONS 'PR (CONS 'GR RIGHT-OUT-NAMES))
               (CONS (CONS 'TV-ALU-HELP (CDR TREE))
                     (CONS 'CL (APPEND RIGHT-A-NAMES
                                       (APPEND RIGHT-B-NAMES
                                               MPGNAMES?)))))
         ;;  P and G
         '((P) (B-AND PL PR))
         '((G) (T-CARRY GL PR GR)))))))))))
```

Figure 10: The PG-ALU Generator, `TV-ALU-HELP*`

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 3: ALU Characteristics

Using the Boyer-Moore theorem prover on a Sun 3/280, the processing of our ALU proof, including the time necessary to define the primitives and interpreters, takes less than one hour. To generate all of the ALU designs for Table 3 only takes a few seconds. A few seconds more are required to check that each ALU is recognized by `NEW-ALU&` and `BOXLIST-OKP`.

10 Translating our HDL into a Commercial CAD Language

One motivation for defining our HDL was to ease the conversion of verified designs into a form acceptable to commercial CAD systems. Instead of attempting to define our own HDL, we could have attempted to formally define an existing HDL, e.g., VHDL [8]. However, before attempting to formalize a complex language, we wanted to explore the subtleties of defining an HDL without the constraints imposed by an existing language. For example, we did not want to spend time worrying over whether we have the "correct" syntax; any unambiguous syntax will do.

Providing a simple translation into commercial HDLs is important for us as we wish to be able to physically realize our verified circuit descriptions. Because our HDL is simple, we expect that designs specified with our HDL should translate to a commercial HDL in a straightforward manner. However, the accuracy of such a translation is open to question. We do not have meaning functions for any commercial HDL; that is, we do not have a function like `HEVAL` for any commercial HDL. Therefore, any translation from our HDL to a commercial HDL cannot be verified.[4]

[4]It seems unlikely that we will have a meaning for a commercial HDL in the near future. Most commercial HDLs that we have investigated are large and complex, and these languages do not have formally defined syntax or semantics. A reasonable approach might be to formally define a subset of a popular commercial HDL. However, even with such a definition the

The Boyer-Moore theorem prover is implemented in Common Lisp and manipulates Lisp representations of Boyer-Moore logic terms. The Boyer-Moore theorem prover provides Common Lisp implementations of functions defined in the logic. These Common Lisp procedures are an integral part of the mechanization of the logic, i.e., the soundness of the theorem prover depends upon their correctness. We use the Common Lisp versions of the formal boxlist-creating functions as the basis of our translations. For example, the boxlist-generating function `(V-ADDER-BOXLIST N)` in the Boyer-Moore logic has a Common Lisp counterpart `(*1*V-ADDER-BOXLIST N)`. Evaluating `(*1*V-ADDER-BOXLIST 4)` in Common Lisp yields the boxlist below.

```
'(((V-ADDER . 4)
   ((CARRY . 5)
    (A . 4) (A . 3) (A . 2) (A . 1)
    (B . 4) (B . 3) (B . 2) (B . 1))

   ((SUM . 4) (SUM . 3) (SUM . 2) (SUM . 1) (CARRY . 1))

   ((((SUM . 4) (CARRY . 4)) (FULL-ADDER (A . 4) (B . 4) (CARRY . 5)))
    (((SUM . 3) (CARRY . 3)) (FULL-ADDER (A . 3) (B . 3) (CARRY . 4)))
    (((SUM . 2) (CARRY . 2)) (FULL-ADDER (A . 2) (B . 2) (CARRY . 3)))
    (((SUM . 1) (CARRY . 1)) (FULL-ADDER (A . 1) (B . 1) (CARRY . 2)))))

  (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)))))
```

To make the translation process concrete, we present the translation of the circuit box above into LSI Logic's NDL. This translation is performed by adding keywords and other bits of syntax to our circuit box expression; translating Lisp lists into comma-separated lists; converting our formal gate names into LSI Logic macrocell names; and translating our CONS-indexed names into a more standard form. The result of converting the boxlist above into NDL is shown in Figure 11.

To perform the translation of boxlists into NDL, we have written a translator in Common Lisp. The complete source text for our translator is less than two pages of Lisp code. Since there is no formal model of NDL, we are not able to verify our translator, but we are pleased that it is so short.

---

software tools used to create integrated circuit masks from HDL circuit specifications will not be verified.

```
COMPILE;
DIRECTORY MASTER;

MODULE V-ADDER_4;
INPUTS CARRY_5,A_4,A_3,A_2,A_1,B_4,B_3,B_2,B_1;
OUTPUTS SUM_4,SUM_3,SUM_2,SUM_1,CARRY_1;
LEVEL FUNCTION;
DEFINE
G_0(SUM_4,CARRY_4) = FULL-ADDER(A_4,B_4,CARRY_5);
G_1(SUM_3,CARRY_3) = FULL-ADDER(A_3,B_3,CARRY_4);
G_2(SUM_2,CARRY_2) = FULL-ADDER(A_2,B_2,CARRY_3);
G_3(SUM_1,CARRY_1) = FULL-ADDER(A_1,B_1,CARRY_2);
END MODULE;

MODULE FULL-ADDER;
INPUTS A,B,C;
OUTPUTS SUM,CARRY;
LEVEL FUNCTION;
DEFINE
G_0(SUM1,CARRY1) = HALF-ADDER(A,B);
G_1(SUM,CARRY2) = HALF-ADDER(SUM1,C);
G_2(CARRY) = OR2(CARRY1,CARRY2);
END MODULE;

MODULE HALF-ADDER;
INPUTS A,B;
OUTPUTS SUM,CARRY;
LEVEL FUNCTION;
DEFINE
G_0(SUM) = EO(A,B);
G_1(CARRY) = AN2(A,B);
END MODULE;

END COMPILE;
END;
```

Figure 11: NDL Translation of a Four-bit Adder

11 Conclusions

The formalization of even a simple, combinational HDL is more subtle and complex than might first be imagined; however, the usefulness of having circuits represented as data cannot be over-emphasized. Our HDL enjoys the benefits of being purpose-built for concisely and directly expressing hardware circuits, but incurs the costs associated with having to prove the correctness of circuit operation through an interpreter.

We believe that it is possible to formalize features of commercial CAD languages. If this is possible, then we may be able to adapt existing verification systems to support commercial CAD systems. We do not expect commercial CAD system suppliers to convert to formal systems (e.g., HOL, Boyer-Moore, NuPRL) without these systems being able to support the kinds of activities commercial customers now enjoy.

The use of arbitrary heuristics is possible for provably correct circuit generating functions. For instance, consider a heuristic that chooses between two correct implementations. The heuristic may perform any amount of analysis, but no matter what choice is returned the resulting circuit generated will still be correct. We employed a simple algebraic choice heuristic in the verification of our ALU generator; this heuristic helps control the fanout of the control lines. It is easy to imagine much more sophisticated heuristics.

Modeling circuits as data in a formal logic has important benefits: well-formed circuits can be formally specified, programs can be used to manipulate circuits, circuits can be verified, and the verification of circuit manipulating tools (e.g., a logic minimizer) can be accomplished. That is, it is possible to build a system where verified circuits are created with the assistance of verified tools. For instance, a verified synthesis program could be constructed, thus eliminating the need for after-the-fact proofs. Our ALU circuit generator is an instance of a verified synthesis program.

We believe the formalization of as much of the design space as possible is critical to the production of higher quality hardware. We are presently extending our HDL to include registers, tri-state and open-collector logic, and other features germane to digital hardware design. It is important that we continue to extend our formal models to include any notion of hardware specification that is important to correctly build computing hardware.

Even though the notion of a formula manual is many years off, we believe formal methods will play an increasingly important role in the construction of predictable computer hardware. The ability to provide multiple interpretations for circuits expressed with our HDL is a step toward providing greater formal modeling coverage of physical properties important to correct hardware device operation.

References

[1] Bishop C. Brock and Warren A. Hunt, Jr. The Formalization of a Simple HDL. In *Proceedings of the IFIP TC10/WG10.2/WG10.5 Workshop on Applied Formal Methods for Correct VLSI Design*, Elsevier Science Publishers, 1990.

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

[3] Geoffrey M. Brown and Miriam E. Lesser. From Programs to Transistors: Verifying Hardware Synthesis Tools. In *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects.*, pages 128–150, Springer Verlag, 1989.

[4] Randal E. Bryant. Verification of Synchronous Circuits by Symbolic Logic Simulation. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects.*, pages 14–24, Springer Verlag, 1989.

[5] Avra Cohn. Correctness Properties of the VIPER Block Model: The Second Level. In *Current Trends in Hardware Verification and Automatic Theorem Proving*, pages 1–91, Springer-Verlag, New York, 1989.

[6] W. J. Cullyer and C. H. Pygott. Application of Formal Methods to the VIPER Microprocessor. *IEE Proceedings*, 134, Pt. E(3):133–141, May 1987.

[7] Diederik Verkest, Luc Claesen, and Hugo De Man. The Use of the Boyer-Moore Theorem Prover for Correctness Proofs of Parameterized Hardware Modules. In *Proceedings of the IFIP TC10/WG10.2/WG10.5 Workshop on Applied Formal Methods for Correct VLSI Design*, Elsevier Science Publishers, 1990.

[8] Editors: Larry Saunders and Ronald Waxman. *IEEE Standard VHDL Language Reference Manual*. The Institute of Electrical and Electronics Engineers, 1988.

[9] Joesph A. Goguen. OBJ as a Theorem Prover with Applications to Hardware Verification. In *Current Trends in Hardware Verification and Automatic Theorem Proving*, pages 218–267, Springer-Verlag, New York, 1989.

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

[11] 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.

[12] Graham Birtwistle, Brian Graham, Todd Simpson, Konrad Slind, Mark Williams, and Simon Williams. Verifying an SECD chip in HOL. In *Proceedings of the IFIP TC10/WG10.2/WG10.5 Workshop on Applied Formal Methods for Correct VLSI Design*, Elsevier Science Publishers, 1990.

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

[14] Steven D. Johnson. Manipulating Logical Organization with System Factorizations. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects.*, pages 259–280, Springer Verlag, 1989.

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

[16] Bill Pace and Mark Saaltink. Formal Verification in m-EVES. In *Current Trends in Hardware Verification and Automatic Theorem Proving*, pages 268–302, Springer-Verlag, New York, 1989.

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

[18] Simon Finn, Michael P. Fourman, Michael Francis, and Robert Harris. Formal System Design - Interactive Synthesis based on Computer-Assisted Formal Reasoning. In *Proceedings of the IFIP TC10/WG10.2/WG10.5 Workshop on Applied Formal Methods for Correct VLSI Design*, Elsevier Science Publishers, 1990.

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

[20] Warren A. Hunt, Jr. and Bishop C. Brock. The Verification of a Bit-Slice ALU. In *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects.*, pages 281–305, Springer Verlag, 1989.