[34] David J. Kuck. *The Structure of Computers and Computations*, John Wiley & Sons: New York, 1978.

[35] Beth H. Levy. An Overview of Hardware Verification using the State Delta Verification System (SDVS), in *Final Program: 1991 International Workshop on Formal Methods in VLSI Design*, January, 1991.

[36] J S. Moore. Verified Hardware Implementing an 8-Bit Parallel I\O Byzantine Agreement Processor. Technical Report 69, Computational Logic, Inc., Austin, Texas, August, 1991.

[37] Larry Saunders and Ronald Waxman, editors. *IEEE Standard VHDL Language Reference Manual*, IEEE, 1988.

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

[39] Mandayam Srivas and Mark Bickford. Formal Verification of a Pipelined Microprocessor, *IEEE Software*, 7:5, September, 1990, pp. 52-64.

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

[41] D.E. Thomas and P. Moorby. The Verilog™ Hardware Description Language, Kluwer Academic Publishers: Boston, 1991.

[42] John Van Tassel and David Hemmendinger. Toward Formal Verification of VHDL Specifications, in *Applied Formal Methods for Correct VLSI Design, Volume 1*, November, 1989, pp. 261-270.

[43] Diederik Verkest, Luc Claesen, Hugo De Man. The Use of the Boyer-Moore Theorem Prover for Correctness Proofs of Parameterized Hardware Modules, in *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, Miriam Leeser and Geoffrey Brown, editors, Springer-Verlag Lecture Notes in Computer Science 408: New York, 1989.

[44] Michael Yoeli, editor. *Formal Verification of Hardware Design*, IEEE Computer Society Press Tutorial: Los Alamitos, CA, 1990.

[22] Warren A. Hunt, Jr. FM8501: A Verified Microprocessor, Technical Report ICSCA-CMP-47, Institute for Computing Science and Computer Applications, University of Texas at Austin, December, 1985.

[23] Warren A. Hunt, Jr. Microprocessor Design Verification, *Journal of Automated Reasoning* 5:4, December, 1989, pp. 429-460.

[24] 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*, Springer Verlag Lecture Notes in Computer Science: New York, 1989, pp. 281-305.

[25] Warren A. Hunt, Jr. and Bishop Brock. A Formal HDL and its use in the FM9001 Verification. *Proceedings of the Royal Society*, to appear April 1992.

[26] 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: Amsterdam, 1990.

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

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

[29] Steven D. Johnson. Manipulating Logical Organization with System Factorizations, in *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, Miriam Leeser and Geoffrey Brown, editors, Springer-Verlag Lecture Notes in Computer Science 408: New York, pp. 259-280, 1989.

[30] Jeffery Joyce, Graham Birtwistle, and M.J.C Gordon. Proving a Computer Correct in Higher Order Logic, Technical Report, University of Calgary, Department of Computer Science, August, 1985.

[31] Matt Kaufmann. A Translator from Brock/Hunt HDL to VHDL, Internal Note 231, May, 1991.

[32] Matt Kaufmann. Some Tools for Using the Brock/Hunt HDL, Internal Note 237, Computational Logic, Inc., Austin, Texas, July, 1991.

[33] Matt Kaufmann. FM92 Survey of Formal Methods: Nqthm and Pc-Nqthm, to appear in *Proceedings of FM92*.

[11] R.E. Bryant. Graph-Based Algoriths for Boolean Function Manipulation, *IEEE Transactions on Computers*, 35:8, August, 1986, pp. 677-691.

[12] Randal E. Bryant. Verification of Synchronous Circuits by Symbolic Logic Simulation, in *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, Miriam Leeser and Geoffrey Brown, editors, Springer-Verlag Lecture Notes in Computer Science 408: New York, pp. 14-24, 1989.

[13] E.M. Clarke, D.E. Long, K.L. McMillan. A Language for Compositional Specification and Verification of Finite State Hardware Controllers, in *IEEE Proc. of the Ninth Symposium on Computer Hardware Description Languages and Their Applications*, 1989, pp. 281-295.

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

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

[16] D.L. Dill and E.M. Clarke. Automatic Verification of Asynchronous Circuits Using Temporal Logic, *Proc. of the IEEE* 133:5, pp. 276-282, September, 1986.

[17] Hans Eveking. Formal Verification of Synchronous Systems, in *Formal Aspects of VLSI Design*, G. Milne and P.A. Subrahmanyam, editors, North-Holland: Amsterdam, 1986, pp. 137-151.

[18] Ivan V. Filippenko. VHDL Verification in the State Delta Verification System, in *Final Program: 1991 International Workshop on Formal Methods in VLSI Design*, January, 1991.

[19] Steven M. German and Karl J. Lieberherr. Zeus: A Language for Expressing Algorithms in Hardware, *IEEE Computer*, 18:2, February, 1985.

[20] Steven M. German and Yu Wang. Formal Verification of Parameterized Hardware Designs, in *Proc. of the International Conference on Computer Design*, 1985.

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

References

[1] R. S. Boyer and J S. Moore. *A Computational Logic*, Academic Press: New York, 1979.

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

[3] William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, William D. Young. An Approach to Systems Verification, *Journal of Automated Reasoning*, 5:4, pp. 411-428, 1989.

[4] 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: Amsterdam, 1990.

[5] Dominique Borrione, Laurence Pierre, and Ashraf Salem. PREVAIL: A Proof Environment for VHDL Descriptions, in *Advanced Research Workshop on Correct Hardware Design Methodologies*, assembled by Politecnico di Torino, Turin, Italy, June, 1991.

[6] Richard Boulton, Mike Bordon, John Herbert, John van Tassel. The HOL Verification of ELLA Designs, in *Final Program: 1991 International Workshop on Formal Methods in VLSI Design*, January, 1991.

[7] 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: Amsterdam, 1990.

[8] Geoffrey M. Brown and Miriam E. Leeser. From Programs to Transistors: Verifying Hardware Synthesis Tools, in *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, Miriam Leeser and Geoffrey Brown, editors, Springer-Verlag Lecture Notes in Computer Science 408: New York, pp. 128-150, 1989.

[9] M. Browne, E.M. Clarke, D.L. Dill, B. Mishra. Automatic Verification of Sequential Circuits Using Temporal Logic, *IEEE Transactions on Computers*, 35:12, December, 1986, pp. 1035-1044.

[10] M. Browne and E.M. Clarke. SML—A High Level Language for the Design and Verification of Finite State Machines, in *From HDL Descriptions to Guaranteed Correct Circuit Designs*, editor D. Borrione, North-Holland: Amsterdam, 1986, pp. 269-292.

```
(CONS (CAR NETLIST) (DELETE-MODULE NAME (CDR NETLIST)))))
```

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

(BVP X)
   =
(IF (NLISTP X)
    (EQUAL X NIL)
  (AND (BOOLP (CAR X)) (BVP (CDR X))))

(F-OR A B)
   =
(IF (OR (EQUAL A T) (EQUAL B T))
    T
  (IF (AND (EQUAL A F) (EQUAL B F))
      F
    (X)))

(F-NOT A) = (IF (BOOLP A) (NOT A) (X))

(F-XOR3 A B C) = (F-XOR A (F-XOR B C))

(XOR A B) = (IF A (IF B F T) (IF B T F))

(B-OR A B) = (OR A B)

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

(B-XOR3 A B C) = (B-XOR (B-XOR A B) C)

(B-AND A B) = (AND A B)

(LOOKUP-MODULE NAME NETLIST)
   =
(IF (NLISTP NETLIST)
    F
  (IF (AND (LISTP (CAR NETLIST))
           (EQUAL (CAAR NETLIST) NAME))
      (CAR NETLIST)
    (LOOKUP-MODULE NAME (CDR NETLIST))))

(DELETE-MODULE NAME NETLIST)
   =
(IF (NLISTP NETLIST)
    NETLIST
  (IF (AND (LISTP (CAR NETLIST))
           (EQUAL (CAAR NETLIST) NAME))
      (CDR NETLIST)
```

- elimination of apparently irrelevant hypotheses; and
- structural induction.

The prover contains heuristics to orchestrate the application of these techniques.

We use the Boyer-Moore theorem prover primarily 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.

The Boyer-Moore system also contains an interpreter for the logic that allows the evaluation of terms in the logic. Thus, the logic can be considered as either a functional programming language or an executable specification language. We often use this facility for debugging our specifications.

The Boyer-Moore prover has been used to check the proofs of some quite deep theorems. For example, some theorems from traditional mathematics that have been mechanically checked using the system include proofs of: the existence and uniqueness of prime factorizations; Gauss' law of quadratic reciprocity; the Church-Rosser theorem for lambda calculus; the infinite Ramsey theorem for the exponent 2 case; and Goedel's incompleteness theorem. Somewhat outside the range of traditional mathematics, the theorem prover has been used to check: the recursive unsolvability of the halting problem for Pure Lisp; the proof of invertibility of a widely used public key encryption algorithm; the correctness of metatheoretic simplifiers for the logic; the correctness of a simple real-time control algorithm; the optimality of a transformation for introducing concurrency into sorting networks; and a verified proof system for the Unity logic of concurrent processes. When connected to a specialized front-end for Fortran, the system has also proved the correctness of Fortran implementations of a fast string searching algorithm and a linear time majority vote algorithm. Recent work at CLI includes the mechanically checked proofs of a high-level language compiler, an assembler, a microprocessor, and a simple multi-tasking operating system. These verified components were integrated into a *vertically verified system* called the "CLI Short Stack"[3], the first such system of which we are aware. Many other interesting theorems have been proven as well. It is important to note that all of these proofs were checked by the same general purpose theorem prover, not a number of specialized routines optimized for specific problems.

Below are definitions of the subsidiary functions used in the examples in the paper.

```
; An  ADD-SHELL event adds a new ``data type''.  In this case, we add
; the new constructor functions X and Z of no arguments, with recognizer
; functions XP and ZP.  This is equivalent to adding to the logic two new
; constants that are distinct from any previously added constants.

    (ADD-SHELL X () XP ())

    (ADD-SHELL Z () ZP ())
```

Appendix:
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 overview of the Boyer-Moore logic and theorem prover.

The Boyer-Moore logic[1, 2] is a quantifier-free, first-order predicate calculus with equality and induction. Logic formulas are written in a prefix-style, Lisp-like notation. Recursive functions may be defined and must be proven to terminate. The logic includes several built-in data types: Booleans, natural numbers, lists, literal atoms, and integers. Additional data types can be defined. The syntax, axioms, and rules of inference of the logic are given precisely in *A Computational Logic Handbook*[2].

The Boyer-Moore logic can be extended by the application of the following axiomatic acts: defining 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 proving system (theorem prover) is a Common Lisp[40] program that provides a user with various commands to extend the logic and to prove theorems. A user enters theorem prover commands through the top-level Common Lisp interpreter. The theorem prover manages the axiom database, function and data type definitions, and proved theorems, thus allowing a user to concentrate on the less mundane aspects of proof development. The theorem prover contains a simplifier and rewriter and decision procedures for propositional logic and linear arithmetic. It also can perform structural inductions automatically.

Theorems are formalized as terms in the logic; the prover attempts to prove a proposed theorem by repeatedly transforming and simplifying it, employing eight basic transformations:

- decision procedures for propositional calculus, equality, and linear arithmetic;
- rewriting based on axioms, definitions and previously proved lemmas;
- automatic application of user-supplied simplification procedures that have been proven correct;
- elimination of calls to certain functions in favor of others that are "better" from a proof perspective;
- heuristic use of equality hypotheses;
- generalization by the replacement of terms by variables;

formula manual (i.e., this collection of formulas) would represent a formal specification that would allow hardware engineers to connect this device with great confidence and allow software engineers to predict the results of programming it.

Formula manuals are still a long way off, but we believe that formalization of our HDL is an incremental step toward our goal. We believe that to address the difficulties of designing, specifying, and constructing complex computing equipment, the hardware verification community must continually expand its modeling efforts to include every possible hardware attribute that contributes 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) other aspects of circuit design, including fanout, loading, and circuit hierarchy and to bring such issues into the realm of formal mathematical modeling.

Acknowledgments: This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406. 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

12 Conclusions

We have embedded within the Boyer-Moore logic a register-transfer level language for describing sequential hardware. Though quite conventional in syntax and semantics, the language and its implementation have a variety of unique aspects.

- The syntax and semantics are fully defined by functions in the computational logic of Boyer and Moore. These functions are executable, permitting effective evaluation of both the combinatorial and sequential aspects of circuit designs.
- Because our HDL is embedded within the Boyer-Moore logic, we have available an expressive formalism for specifying properties of our circuit designs.
- Since our formalism is supported by a powerful heuristic theorem prover, we can establish with mathematical rigor the conformance of our circuit designs to their specifications.
- Our circuit interpreters compute various important aspects of circuit behavior, such as fanout, in addition to functional results;
- Since circuits are represented by logical constants, we can manipulate them via functions in the logic. This opens the possibility of constructing verified tools, such as minimizers, tautology checkers, etc., that manipulate circuit expressions.
- We are able to construct and prove parameterized circuit generator functions, allowing automatic synthesis of proven circuits.
- The language is semantically akin to commercial HDL's; this permits mechanical translation to a form readily processed by available commercial tools.

Our long-term vision for hardware verification research is to eventually see a much more formal approach to hardware design than the current practice. We envision a standard style of formal specification for digital hardware that we call a *formula manual*. A formula manual is a fully formal description of a device that records *all* of the information that a manufacturer would need to fabricate the device and a user would need to use it reliably. For instance, a microprocessor's formula manual would contain a series of mathematical formulas that describe the programming model, the timing diagrams, the memory interface, the pin-out, the power requirements, cooling requirements, etc. The lowest level mathematical model of the implementation of this microprocessor would be verified to meet every specification given by the formula manual. Then the

```
T2(CARRY)=OR2(CARRY1,CARRY2);        (T2 (CARRY) B-OR (CARRY1 CARRY2)))
                                     NIL)
END MODULE;

MODULE HALF-ADDER;                  (HALF-ADDER
INPUTS A,B;                          (A B)
OUTPUTS SUM,CARRY;                   (SUM CARRY)
LEVEL FUNCTION;
DEFINE
G0(SUM) = E0(A,B);                   ((G0 (SUM) B-XOR (A B))
G1(CARRY) = AN2(A,B)                  (G1 (CARRY) B-AND (A B)))
                                     NIL))
END MODULE;
```

... four additional modules (not shown) ...

```
END COMPILE;
END;
```

Notice that the modules `FULL-ADDER` and `HALF-ADDER` are minor syntactic variants of our HDL descriptions. We translate our primitive names into the corresponding NDL primitives. Additional modules (not shown) are automatically added by the translator to provide circuitry to facilitate testing, and primitive modules for signal renaming, logical `TRUE`, and logical `FALSE` for the circuit. Given this translation, we can also process the NDL description of the module with LSI Logic's "schematic liberator" program and obtain a mechanically drawn schematic diagram for our circuit.

Each of our circuit primitives has a corresponding NDL counterpart. Consequently, all HDL circuits are translatable into NDL, and hence implementable with accessible technology. We cannot exhibit an NDL display of a parameterized module such as those described by our circuit generator functions; but we can translate any specific instance of them. We take our ability to translate our formal descriptions into a commercial HDL as strong evidence that our HDL provides a reliable basis for circuit implementation. The translation from our HDL to NDL is not formally verified in any sense, and could not be because NDL does not have a formal semantics. However, we have considerable evidence from comparing the results of the LSI simulator and our `DUAL-EVAL` function that the semantics of our hardware primitives coincides with their NDL counterparts.

In addition to our translator to NDL, we have also constructed translation programs from our HDL to the MIMIC hardware description language[32] and to VHDL[31].

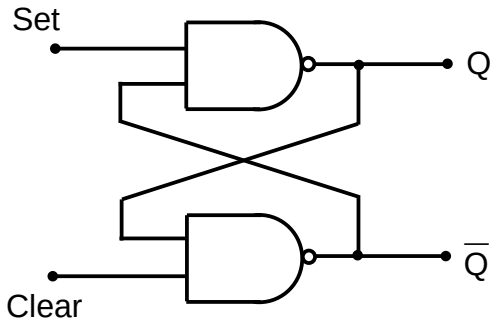


Figure 7: NAND-Latch Not Handled by our HDL

11 Translation to the LSI Logic Form

A hardware description language is useful only to the extent that it can be used for expressing *implementable* hardware designs. Our HDL is designed as a formal abstraction of a sequential subset of a generic commercial CAD language. We believe that a netlist 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 our formal circuit descriptions.

We have written a Common Lisp program that translates our HDL circuit descriptions into LSI Logic's Netlist Description Language (NDL). NDL is a conventional hardware description language similar to Verilog™[41]. Commercially available tools from LSI Logic permit one to analyze NDL descriptions to extract schematics, do layout, etc. We used our translator to generate NDL netlists from our HDL description of the FM9001 microprocessor[25]. These netlists were delivered to LSI Logic and were the basis for the fabrication of the FM9001.

The translation is extremely staightforward; the translator itself occupies around a single page of Common Lisp code. Below is the `FULL-ADDER` netlist from Section 4, juxtaposed with its translation to NDL.

```
COMPILE;
DIRECTORY MASTER;

MODULE FULL-ADDER;                '((FULL-ADDER
INPUTS A,B,C;                        (A B C)
OUTPUTS SUM,CARRY;                   (SUM CARRY)
LEVEL FUNCTION;
DEFINE
T0(SUM1,CARRY1)=HALF-ADDER(A,B);    ((T0 (SUM1 CARRY1) HALF-ADDER (A B))
T1(SUM,CARRY2)=HALF-ADDER(SUM1,C);   (T1 (SUM CARRY2)  HALF-ADDER (SUM1 C))
```

Primitive	Description
AO2, AO4, AO6, AO7:	OR-AND circuits;
B1I:	clock buffer;
B-AND, B-AND3, B-AND4:	bitwise AND;
B-EQUV, B-EQUV3:	bitwise exclusive NOR;
B-IF:	bit selector function;
B-NAND, B-NAND3, B-NAND4, B-NAND5, B-NAND6, B-NAND8:	bitwise NAND;
B-NBUF:	buffer;
B-NOR, B-NOR3, B-NOR4, B-NOR5, B-NOR6, B-NOR8:	bitwise NOR;
B-NOT:	bitwise NOT;
B-NOT-B4IP:	high power inverter;
B-NOT-IVAP:	high power inverter;
B-OR, B-OR3, B-OR4:	bitwise OR;
B-XOR, B-XOR3:	bitwise exclusive OR;
DEL2, DEL4, DEL10:	delay elements;
PROCMON:	LSI process monitor
DP-RAM-16X32:	16 word $\times$ 32-bit register file;
FD1, FD1S, FD1SP, FD1SLP:	D flip-flops;
ID:	wire renaming primitive;
MEM-32X32:	32-bit memory element;
RAM-ENABLE-CIRCUIT:	level-sensitive memory enable;
T-BUF:	tri-state buffer;
T-WIRE:	tri-state bus element;
PULLUP:	tri-state or open collector pullup;
TTL-BIDIRECT:	bi-directional I/O pad;
TTL-CLK-INPUT:	clock input pad;
TTL-INPUT:	input line;
TTL-OUTPUT-PARAMETRIC:	ttl parametric output;
TTL-OUTPUT, TTL-OUTPUT-FAST:	output lines;
TTL-TRI-OUTPUT, TTL-TRI-OUTPUT-FAST:	tri-state output line;
VDD, VDD-PARAMETRIC:	power;
VSS:	ground;

Different types of a given primitive indicate either different input arities (eg. B-NOR, B-NOR3, etc) or different gate types (FD1, FD1S, etc).

Figure 6: Currently Defined Hardware Primitives

10 Admissible Circuits

Our HDL provides a way to specify, verify, and compose sequential machines. The degenerate case is modules with no state-holding devices, i.e., purely combinational. The class of circuits acceptable to our tools is formally defined by a collection of recognizer predicates written in the Boyer-Moore logic. The predicates that recognize a well-formed netlist check to see that each module it contains is syntactically correct—that it contains no combinational loops or wire type mismatches, that it has correct clock distribution, that it contains no overloading of any primitive, that any reference to a state-holding device is listed in the state list argument, that the arities of all module and primitive references are correct, etc. All acceptable netlists are lists of module definitions that obey these constraints and reference only primitives or other defined modules.

Separately, we check the consistency of our database of information about the primitive elements to ensure that every defined primitive has information about its delays, inputs types, input loadings, output types, output drive strengths, and uniqueness of all port names. Figure 6 gives our currently defined list of primitives. For combinational primitives we check that each primitive has a database entry that specifies its combinational behavior; for sequential primitives we check that the database defines both a combinational result and a means to compute the next state for the primitives internal state-holding devices.

There are certain limitations to the expressive power of our HDL. The treatment of time in our interpreter model is somewhat simplistic. Our hardware description language only admits synchronous sequential circuits with a single (implicit) clock. A call to `DUAL-EVAL` is assumed to model one "tick" of the global clock. All transitions are triggered by the leading edge of the clock pulse. All state-holding devices update their internal states simultaneously. This prevents us from modeling some simple circuits such as the cross-coupled `NAND` latch shown in Figure 7. Though obviously less general than an event-driven simulator model such as that of VHDL[37], this restriction makes the modeling and proof problem much more tractable and guarantees that our circuit descriptions can be composed. The fact that the clock is implicit restricts the class of circuits we can handle and the properties we can specify of them. For example, we do not allowed gated-clock circuits, though it would require only a minor extension to the language to do so.

`DUAL-EVAL` is essentially a unit-delay symbolic simulator. The restrictions we impose are designed to ensure that all circuit evaluations terminate deterministically. Similar restrictions are imposed, for example, in [9] who "believe that verification under the unit-delay assumption is a good way to debug many types of asynchronous circuits—perhaps as an initial step in a more thorough (and expensive) verification process." We chose these restrictions so as to not sacrifice the benefits of hierarchical design and proof.

Notice that this is exactly the form of the hand-coded 4-bit adder from Section 4. However, the function `RIPPLE-ADDER*` is a circuit generator that will produce an adder of arbitrary length. Consequently, proving the correctness of this parameterized generator function gives us a proof of an infinite class of circuits, any of which can be generated automatically. We have used such circuit generator functions extensively in defining the ALU for the FM9001, various register files, n-bit selectors, and other hardware modules.

9 Proofs of Circuit Generators

We desire to show that the circuits produced by our generator functions are correct. For the n-bit adder generator described in the previous section, this means that the generated circuit produces the appropriate $n+1$ bit sum of two n-bit strings. This is stated in the following theorem.

Theorem. `RIPPLE-ADDER$Value`.

```
(IMPLIES
 (AND (RIPPLE-ADDER& NETLIST N)
      (BOOLP C) (BVP A) (BVP B)
      (EQUAL (LENGTH A) N)
      (EQUAL (LENGTH B) N))
 (EQUAL (DUAL-EVAL 'OUTPUTS (CONS 'RIPPLE-ADDER N)
                   (CONS C (APPEND A B)) STATE NETLIST)
        (BV-ADDER C A B))).
```

We can interpret this theorem as follows: the result of evaluating the circuit module generated by `(RIPPLE-ADDER* N)` on parameters `(CONS C (APPEND A B))`, `STATE`, and `NETLIST` will be equal to the result of executing the defined recursive function `BV-ADDER` on parameters `C`, `A`, and `B`, provided the following hold:

- `NETLIST` is a netlist defining the circuit result of `(RIPPLE-ADDER* N)`; and
- `C` is a Boolean; `A` and `B` are bit-vectors of length `N`.

Notice that this is generalization of our earlier theorem $\texttt{RIPPLE-ADDER}_4\texttt{\$Value}$. Whereas the 4-bit case could be proved by flattening of the gate graph or by exhaustive simulation, the proof of the general case requires induction. The proof of `RIPPLE-ADDER$Value` is by induction on the recursive structure of `RIPPLE-ADDER-BODY`. Proofs of such theorems can be rather difficult and may require considerable familiarity with the structure of the interpreter and with the Boyer-Moore prover. One of our continuing goals is to simplify this aspect of using the methodology.

```
(RIPPLE-ADDER-BODY M N)
   =
(IF (ZEROP N)
    NIL
    (CONS (LIST (INDEX (G M)
                (LIST (INDEX (SUM M)
                      (INDEX (CARRY (ADD1 M)))
                'FULL-ADDER
                (LIST (INDEX (A M)
                      (INDEX (B M)
                      (INDEX (CARRY M)))
          (RIPPLE-ADDER-BODY (ADD1 M) (SUB1 N))))
```

Shown below is the result of executing `(RIPPLE-ADDER-BODY 0 4)`.

```
'((G0 (SUM0 CARRY1) FULL-ADDER (A0 B0 CARRY0))
  (G1 (SUM1 CARRY2) FULL-ADDER (A1 B1 CARRY1))
  (G2 (SUM2 CARRY3) FULL-ADDER (A2 B2 CARRY2))
  (G3 (SUM3 CARRY4) FULL-ADDER (A3 B3 CARRY3)))
```

To complete our ripple-adder generator function, we insert the `RIPPLE-ADDER-BODY` into a circuit module with appropriate input and output names; the function `RIPPLE-ADDER*` produces such a circuit description. The function call `(GENERATE-NAMES NAME N)` produces the list of names $\texttt{NAME}_0$, $\texttt{NAME}_1$, ..., $(\texttt{NAME}_{N-1})$.

```
(RIPPLE-ADDER* N)
   =
(LIST (CONS 'RIPPLE-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)))
      (RIPPLE-ADDER-BODY 0 N)
      NIL))
```

Evaluating `(RIPPLE-ADDER* 4)` gives:

```
'(RIPPLE-ADDER4
   (CARRY0 A0 A1 A2 A3 B0 B1 B2 B3)
   (SUM0 SUM1 SUM2 SUM3 CARRY4)
   ((G0(SUM0 CARRY1) FULL-ADDER (A0 B0 CARRY0))
    (G1(SUM1 CARRY2) FULL-ADDER (A1 B1 CARRY1))
    (G2(SUM2 CARRY3) FULL-ADDER (A2 B2 CARRY2))
    (G3(SUM3 CARRY4) FULL-ADDER (A3 B3 CARRY3)))
   NIL)
```

circuits descriptions 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. We illustrate our approach in this section by returning to our example of an n-bit, ripple-carry adder and showing how to write and verify a generator function that will produce a correct adder for arbitrary word size. The proofs of generator functions are discussed in Section 9.

Recall from Section 4, that a 4-bit ripple-carry adder can be defined in our HDL as follows:

```
'(RIPPLE-ADDER4
   (CARRY0 A0 A1 A2 A3 B0 B1 B2 B3)
   (SUM0 SUM1 SUM2 SUM3 CARRY4)
   ((G0(SUM0 CARRY1) FULL-ADDER (A0 B0 CARRY0))
    (G1(SUM1 CARRY2) FULL-ADDER (A1 B1 CARRY1))
    (G2(SUM2 CARRY3) FULL-ADDER (A2 B2 CARRY2))
    (G3(SUM3 CARRY4) FULL-ADDER (A3 B3 CARRY3)))
   NIL)
```

We can clearly define such a constant for an n-bit adder for any fixed n. In general, an n-bit circuit description would be written as follows.

```
'(RIPPLE-ADDERn
  (CARRY0 A0 ... An-1 B0 ... Bn-1)
  (SUM0 ... SUMn-1 CARRYn)
  ((G0(SUM0 CARRY1) FULL-ADDER (A0 B0 CARRY0))
                    :
   (Gn-1(SUMn-1 CARRYn) FULL-ADDER (An-1 Bn-1 CARRYn-1))))
```

However, rather than constructing an explicit circuit description for particular values of n, we take a more general approach. We define within the Boyer-Moore logic a function that creates an adder circuit module, where the width is provided as a parameter to the function.

We construct our ripple-carry adder module generator in two parts: a top-level function provides the module name, the input names, and the output names; an auxiliary function creates the list of `FULL-ADDER` occurrences that make up the ripple-carry adder body. The function `(RIPPLE-ADDER-BODY M N)`, shown below, creates a list of `N` occurrences of `FULL-ADDER` with appropriate connections to the adjacent occurrences, numbering from `M`.

Consider the theorem `FULL-ADDER$Value` from the previous section. This theorem provides an explicit value of an evaluation of module `FULL-ADDER` under fairly general circumstances. Consequently, a symbolic evaluation of any call to a module that uses `FULL-ADDER` need not expand the definition. Rather, it can use theorem `FULL-ADDER$Value` as a *rewrite rule* to give the output values without delving into the definition of `FULL-ADDER`, provided that the hypotheses of the theorem are satisfied in that context. The proof of `FULL-ADDER`, in turn, uses the `HALF-ADDER$Value` lemma to characterize the results of the two calls to `HALF-ADDER` without having to open them up. At the lowest level, we provide lemmas explicitly characterizing the results of each of the primitives. Then, at each level of the design hierarchy, the results from lower levels are encapsulated in the form of lemmas. We do this for both the module outputs and next state. There is never any need to flatten hierarchically structured designs for proof purposes.

This suggests our general strategy for parallel design and proof. Each of the hardware primitives is defined to return an explicit result. Whenever a module is defined, the user characterizes the results of a call to that module in terms of a recursive function of the inputs and current values of the state-holding components, and proves the corresponding theorem as a Boyer-Moore rewrite rule. Subsequent module definitions using the defined module can rely upon its result, as characterized by the correctness theorem. Subsequent proofs use these theorems to provide the module output and state values without needing to unfold the definition. Following this approach, proofs of very complex modules can be constructed with the exact hierarchical structure used in the design. Proof complexity is "essentially" linear in the number of distinct module types, rather than in the number of occurrences of primitives.

Though we have suggested that the strategy is straightforward, considerable familiarity with the Boyer-Moore theorem prover is still needed to prove complex modules. For example, defining the appropriate recursive functions to characterize results of circuit evaluation is not always easy. Care is required to make the theorems as general as possible. Certain function definitions must be "disabled" to prevent them from opening up during proofs. We are still refining our methodology and attempting to automate some of the more mundane aspects. However, we have been able to use the methodology to prove the correctness of a number of devices, including a 32-bit microprocessor[25], an 8-bit Byzantine resilient processor[36], and a simple combination lock.

8 Circuit Generator Functions

In addition to defining circuit modules as explicit Boyer-Moore constants, we can construct functions to generate these constants. We then prove properties about the circuit generator functions and, thus, about the resulting circuit modules. The purpose of circuit generator functions is to create parameterized "generic"

Theorem. Latch-State-also-Boolean.

```
(IMPLIES
 (AND (M2& NETLIST)
      (BOOLP D0) (BOOLP D1)
      SEL0 SEL1
      (BOOLP SEL0) (BOOLP SEL1))
 (BVP (DUAL-EVAL 'STATE 'M2 (LIST CLK EN0 EN1 SEL0 SEL1 D0 D1)
                 (LIST S0 S1) NETLIST)))
```

This example suggests the value of our approach. Using simulation it would be possible to exhaustively prove the statements shown above. But to state general theorems of this type and later make use these theorem requires a more general approach. We often combine circuits like `M2` into circuits too large for exhaustive simulation.

In addition to these sorts of "behavioral" properties of circuits, we can express "structural" properties of our modules. For example, it is easy to express (and prove) that no module in a circuit description exceeds certain fanout limitations. This provides some additional assurance that our circuit designs can be implemented within a given technology.

7 Proofs of Circuits

Our circuit design primitives are described at the register-transfer level. However, it should be apparent that we can use our primitives and defined modules hierarchically to design more and more complex modules. We saw, for example, in the previous sections how our `FULL-ADDER` module was built up from primitives and the defined module `HALF-ADDER`. In turn we used `FULL-ADDER` in the definition of $\texttt{RIPPLE-ADDER}_4$, and so on.

Any complex module defined hierarchically can ultimately be "flattened" into an elaborate graph of primitives, since there is no recursion in our definitions. However, to do so sacrifices many of the benefits of hierarchical design. We would like to maintain the benefits of hierarchical structure both for design *and for proof*. This contrasts with some other hardware verification methodologies that flatten a design and apply brute force Boolean procedures[11]. We would like to make use of hierarchical structure both at design and at proof time. Our methodology allows this.

Theorem. M2$State.

```
(IMPLIES
 (M2& NETLIST)
 (EQUAL (DUAL-EVAL 'STATE 'M2
                   (LIST CLK EN0 EN1 SEL0 SEL1 D0 D1) STATE NETLIST)
        (LET ((Q (FT-WIRE (FT-BUF EN0 (CAR STATE))
                          (FT-BUF EN1 (CADR STATE)))))
          (LIST (F-IF SEL0 D0 Q) (F-IF SEL1 D1 Q)))))
```

In addition to these general lemmas characterizing the output and state values of our modules, we often require more specialized lemmas about the results of evaluating our circuit module constants. For example, it does not follow from our syntactic checks alone, that the state values of `M2` remain Boolean. In our model, if a Boolean latch attempts to store a non-Boolean value, `DUAL-EVAL` produces a new state value of `(X)` or "undefined." It is possible, however, to prove that the latches in `M2` will always latch a Boolean value, given certain conditions. If `SEL0` is `T` then only input `D0` need be Boolean. If `SEL0` is `F` then the value on the bus must resolve to a Boolean value. Similar remarks apply to `SEL1`, `D1`, and the bus result. This is expressed formally in the theorem below.

Theorem. Latch-State-Boolean.

```
(IMPLIES
 (AND (M2& NETLIST)
      (BOOLP SEL0) (BOOLP SEL1)
      (LET ((Q (FT-WIRE (FT-BUF EN0 S0)
                        (FT-BUF EN1 S1))))
        (AND (IF SEL0 (BOOLP D0) (BOOLP Q))
             (IF SEL1 (BOOLP D1) (BOOLP Q)))))
 (BVP (DUAL-EVAL 'STATE 'M2 (LIST CLK EN0 EN1 SEL0 SEL1 D0 D1)
                 (LIST S0 S1) NETLIST)))
```

Notice that we require `SEL0` and `SEL1` to be Boolean; otherwise, the `F-IF` primitive will produce an undefined result even if both of its inputs are Boolean.

A variation of this theorem is sometimes useful. If both select lines are high, then we only need to know that the data inputs are Boolean and that the select lines are set to select data inputs `D0` and `D1`, to assure that the state values are Boolean.

Unlike our previous examples, associated with module M1 is a lemma that describes its next state function.

```
Theorem. M1$State.
 (IMPLIES
  (M1& NETLIST)
  (EQUAL (DUAL-EVAL 'STATE 'M1 (LIST CLK EN SEL D Q) STATE NETLIST)
         (F-IF SEL D Q)))
```

M1$State is a lemma stating that the value of the next state of the one-bit latch contained in M1 is (F-IF SEL D Q). We can use both the next state and the value lemmas for modules that contain references to module M1.

Similarly, we define and prove output and next state lemmas for module M2. Module M2 references two M1 modules and T-WIRE, our bus resolution primitive. Function FT-WIRE describes the behavior of T-WIRE.

```
(FOURP X) = (OR (EQUAL X T) (EQUAL F X) (EQUAL X (X)) (EQUAL X (Z)))

(FOURFIX X) = (IF (FOURP X) X (X))

(FT-WIRE A B)
   =
(IF (EQUAL A B) (FOURFIX A)
  (IF (EQUAL A (Z)) (FOURFIX B)
    (IF (EQUAL B (Z)) (FOURFIX A)
      (X))))
```

The following two lemmas characterize the operation of module M2. Note that lemma M2$State shows the next state of M2 to be a list containing two elements. Recall from Section 4 that the state component of our M2 module was a list of two elements, corresponding to the states of the two component occurrences of module M1.

```
Theorem. M2$Value.
 (IMPLIES
  (M2& NETLIST)
  (EQUAL (DUAL-EVAL 'OUTPUTS 'M2
                    (LIST CLK EN0 EN1 SEL0 SEL1 D0 D1) STATE NETLIST)
         (LIST (FT-WIRE (FT-BUF EN0 (CAR STATE))
                        (FT-BUF EN1 (CADR STATE))))))
```

- NETLIST is an appropriate netlist defining the circuit RIPPLE-ADDER$_4$.
- The various arguments are bit lists of the requisite lengths.

BV-ADDER is a recursively defined specification function that computes the bitwise sum of two bit vectors. Notice that though BV-ADDER is defined in a very structural hardware-oriented fashion, its result is provably equal to the arithmetic $n+1$-bit sum of A, B, and C when these are interpreted as binary numbers of appropriate lengths. Thus, our 4-bit adder circuit adds, as desired.

```
(BV-ADDER C A B)
   =
(IF (NLISTP A)
    (LIST C)
  (LET ((A-IN (CAR A))
        (B-IN (CAR B)))
    (CONS (XOR C (XOR A-IN B-IN))
          (BV-ADDER (OR (AND A-IN B-IN)
                        (AND A-IN C)
                        (AND B-IN C))
                    (CDR A)
                    (CDR B)))))
```

As a final example, we consider the specification of properties of sequential logic of the previously presented modules M1 and M2, recognized in a netlist by predicates M1& and M2&. For modules with state-holding components we define lemmas characterizing both the module's outputs and the module's next state. The following lemma describes the outputs of module M1.

Theorem. M1$Value.

```
(IMPLIES
 (M1& NETLIST)
 (EQUAL (DUAL-EVAL 'OUTPUTS 'M1 (LIST CLK EN SEL D Q) STATE NETLIST)
        (LIST (FT-BUF EN STATE))))
```

Function FT-BUF is used to characterize the properties of primitive T-BUF.

```
(FT-BUF C A)
   =
(IF (EQUAL C T) (THREEFIX A)
  (IF (EQUAL C F) (Z) (X)))

(F-IF C A B)
   =
(IF (EQUAL C T) (THREEFIX A)
  (IF (EQUAL C F) (THREEFIX B)
    (X)))
```

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

(THREEFIX A) = (IF (BOOLP A) A (X))

(F-XOR A B)
   =
(IF (EQUAL A T) (F-NOT B)
  (IF (EQUAL A F) (THREEFIX B)
    (X)))
```

Our four-valued logic functions return `(X)` if given unknown inputs. For example, note that `F-AND` only returns a Boolean result if one of its inputs is `F` or both of its inputs are `T`; otherwise, it returns `(X)`. Thus, both the input and output of `DUAL-EVAL` can include `(Z)` and `(X)` values, as well as Booleans. For particular circuits, we often prove the general theorem about the four-valued behavior of the circuit and then the restriction to the Boolean-valued case.[7]

As another example, consider our 4-bit ripple-adder module. We desire to prove that it produces the appropriate 5-bit sum of two 4-bit strings. This is stated in the following theorem.

Theorem. RIPPLE-ADDER$_4$$Value.

```
(IMPLIES
 (AND (RIPPLE-ADDER4& NETLIST)
      (BOOLP C)
      (BOOLP A0) (BOOLP A1) (BOOLP A2) (BOOLP A3)
      (BOOLP B0) (BOOLP B1) (BOOLP B2) (BOOLP B3))
 (EQUAL (DUAL-EVAL 'OUTPUTS 'RIPPLE-ADDER4
                   (LIST CARRY0 A0 A1 A2 A3 B0 B1 B2 B3)
                   STATE NETLIST)
        (BV-ADDER C (LIST A0 A1 A2 A3)
                    (LIST B0 B1 B2 B3)))).
```

The interpretation of this theorem is as follows: the result of evaluating the circuit module `RIPPLE-ADDER`$_4$ on the bit lists `A` and `B`, and Boolean `C` with respect to `STATE` and `NETLIST` will be equal to the result of executing the defined recursive function `BV-ADDER` on those bit lists provided that the following hold.

[7] Sometimes it is crucial to consider the non-Boolean case. For example, to prove the correctness of a sequential circuit's reset behavior, the circuit must be designed to trap the unknown values that exist upon "powering up" the hardware. For this purpose we use primitives like `B-AND` that provide a way for us to force the output to a known value in face of unknown inputs. Certain gates "trap" unknown values; other gates such as `B-XOR` produce the unknown value `(X)` if either of their inputs are not Boolean.

As an example of our specification style, we can express the desired value of a call to FULL-ADDER as an appropriate Boolean expression and prove that, under certain assumptions, the module always returns that value. Formally, the following is a theorem in the logic:

```
Theorem.  FULL-ADDER$Value.
 (IMPLIES (AND (FULL-ADDER& NETLIST)
               (BOOLP C) (BOOLP A) (BOOLP B))
          (EQUAL (DUAL-EVAL 'OUTPUTS 'FULL-ADDER
                            (LIST C A B) STATE NETLIST)
                 (FULL-ADDER-B-VALUE C A B))),
```

where we define the function FULL-ADDER-B-VALUE as follows:

```
(FULL-ADDER-B-VALUE C A B)
   =
(LIST (B-XOR3 C A B)
      (B-OR (B-AND A (B-OR B C))
            (B-AND B C)).
```

The theorem asserts that if NETLIST is appropriately structured to include the definition of FULL-ADDER and its subsidiary modules, and the three inputs are all Boolean valued, then the outputs have their intended values.

Actually, the theorem FULL-ADDER$Value above is a specialization of a more general theorem that defines the results for arbitrary (including non-Boolean) values of the inputs. The more general theorem value theorem for FULL-ADDER is below, though we usually require only the Boolean-valued version.

```
Theorem.  FULL-ADDER-FOUR-VALUED$Value.
 (IMPLIES (FULL-ADDER& NETLIST)
          (EQUAL (DUAL-EVAL 'OUTPUTS 'FULL-ADDER
                            (LIST C A B) STATE NETLIST)
                 (FULL-ADDER-VALUE C A B))).
```

Here, FULL-ADDER-VALUE is the corresponding generalization of FULL-ADDER-B-VALUE:

```
(FULL-ADDER-VALUE C A B)
   =
(LIST (F-XOR3 C A B)
      (F-OR (F-AND A (F-OR B C))
            (F-AND B C))).
```

Functions such as F-XOR and F-AND are the four-valued counterparts of the standard Boolean functions. For example, we define F-AND and F-XOR as follows.

The examples above have all assumed Boolean values. However, the underlying evaluation model used by DUAL-EVAL is four-valued, with logical values of T, F, (Z), and (X); (Z) is the "floating value and (X) is "undefined."[6] All of the primitives (and, hereditarily, all defined modules) are defined on the four logical values. For instance, the call

```
(DUAL-EVAL 'OUTPUTS 'M2 (LIST T F F T F T T) (LIST F F) M2-NETLIST)
```

produces (LIST (Z)), since the two tri-state enables (the second and third elements of the *arguments*) are both off. If both enable bits are on and the state-holding components are different, the value for

```
(DUAL-EVAL 'OUTPUTS 'M2 (LIST T T T T F T T) (LIST T F) M2-NETLIST)
```

will be (LIST (X)), because of the bus conflict. The state-holding components can also become (X) if they attempt to latch a value of (Z). For this circuit, a clock argument is syntactically required, but it plays no part in the evaluation.

The values produced by DUAL-EVAL are computed by a recursive evaluation of a module definition and its component modules. The syntactic constraints on the language assure that this can be done efficiently, in two passes through the occurrence list. Since all of our functions, including DUAL-EVAL, are defined within the Boyer-Moore logic, we can use the logic's interpreter to simulate them for explicit input values. We use this capability extensively to "debug" our efforts before investing the time to prove the correctness of our circuit designs.

The treatment of time in our interpreter model is simplistic. Our "unit clock" simulator is defined by the recursive application of DUAL-EVAL. Potential extensions of our model are described in Section 12.

6 Specifying Hardware Properties

Our DUAL-EVAL interpreter provides an operational semantics for our circuit description language. Because it is defined as a function in the Boyer-Moore logic, we can use the logic to express desired properties of our modules and netlists. We chose to semantically embed a language in the Boyer-Moore logic because we wanted an HDL that was specifically organized to express hardware circuits. Our HDL has been designed for describing hardware circuits and their structure, providing a clear interface between what we consider a hardware description and what is just a Boyer-Moore logic definition.

[6] (Z) and (X) are predefined zero-ary (constant) functions. T and F are abbreviations for the constant functions (TRUE) and (FALSE).

A *flag* value of `'OUTPUTS` or `'STATE` determines, respectively, whether the interpreter computes the values of the output lines or the new values of the state-holding components. The other arguments are:

- *function*: the name of the module to be evaluated;
- *arguments*: the list of values of the input lines for this module evaluation;
- *state*: a list of current values of the state-holding components of the module; and
- *netlist*: the netlist in which module *function* is defined.

For example, assuming that `NETLIST` is the netlist for `FULL-ADDER` given in the previous section and `STATE` is any Boyer-Moore object[5], the function call:

```
(DUAL-EVAL 'OUTPUTS 'FULL-ADDER (LIST T F T) STATE NETLIST)
```

returns

```
(LIST F T).
```

This returned value represents the list of values of the `FULL-ADDER` outputs `SUM` and `CARRY`, when the inputs `A`, `B`, and `C` take values `T`, `F`, and `T`, respectively. The call

```
(DUAL-EVAL 'STATE 'FULL-ADDER (LIST T F T) STATE NETLIST),
```

returns the list of updated values of the state-holding components of module `FULL-ADDER` on the same arguments, and in this case returns `NIL` since the `FULL-ADDER` module has no state-holding components. The call

```
(DUAL-EVAL 'STATE 'M2 (LIST T T T T F T T) (LIST F F) M2-NETLIST)
```

returns `(LIST T F)`, the list of values of the two state-holding components of module `M2` under the given assignment of values to the input lines and previous values to the state-holding components. The argument `M2-NETLIST` must be an appropriate netlist containing the definitions of `M1` and `M2` given in Section 4.

[5]You might expect that the *state* argument should be `NIL`, since `FULL-ADDER` has no state-holding components. However, `DUAL-EVAL` merely tries to bind the names of each of the state-holding arguments of the module—in this case there are none—to whatever values appear in `STATE`. Any additional structure of `STATE` is simply ignored.

As a final example of a module definition in our HDL, consider a 4-bit ripple-carry adder constructed by connecting four FULL-ADDER modules. Such an adder is illustrated in Figure 5, where A_3 is the most significant bit and the adder carries from right to left. The circuit module HDL constant for this adder is shown below.[4]

```
'(RIPPLE-ADDER4
  (CARRY0 A0 A1 A2 A3 B0 B1 B2 B3)
  (SUM0 SUM1 SUM2 SUM3 CARRY4)
  ((G0(SUM0 CARRY1) FULL-ADDER (A0 B0 CARRY0))
   (G1(SUM1 CARRY2) FULL-ADDER (A1 B1 CARRY1))
   (G2(SUM2 CARRY3) FULL-ADDER (A2 B2 CARRY2))
   (G3(SUM3 CARRY4) FULL-ADDER (A3 B3 CARRY3)))
  NIL)
```

In section 8, we will show how we can construct and verify a *generator function* to construct a correct instance of this module for arbitrary word size.

The various syntactic constraints on our circuit descriptions described informally above are checked by a collection of functions (predicates) written in the Boyer-Moore logic. We have defined predicates to check for well-formed names, the absence of combinational loops, loading and fanout violations, wire type mismatches, clock distribution, and other circuit parameters. These predicates are described further in Section 10.

5 Hardware Interpreters

A module in our HDL is a representation of a hardware circuit design. Conceptually, the *meaning* of the module is the computation performed by that circuit for given inputs. This is formalized operationally by a collection of interpreter functions that *evaluate* circuit modules for specific values of the input variables, and produce the appropriate values for the output variables and state-holding variables. These interpreter functions give meaningful values only for well-formed circuit descriptions. Our syntactic constraints assure that a well-formed circuit module body is ordered such that the outputs can be computed efficiently by our interpreters.

The operational semantics of our circuit description language is given by an interpreter function DUAL-EVAL coded in the Boyer-Moore logic. A call to DUAL-EVAL takes the form:

(DUAL-EVAL *flag function arguments state netlist*).

[4] In this paper we use the notational convention X_i, where we would actually be forced in the logic to write XI, or sometimes (INDEX X I).

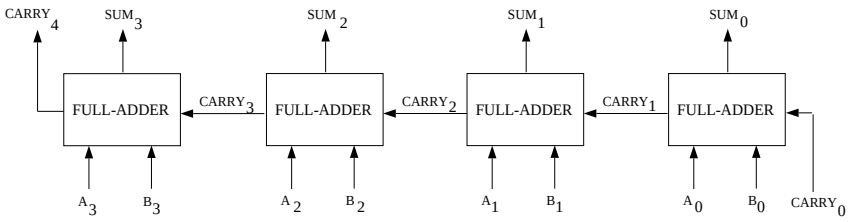


Figure 5: Four-bit, Ripple-carry Adder Circuit

```
'((FULL-ADDER (A B C) (SUM CARRY)
              ((T0 (SUM1 CARRY1) HALF-ADDER (A B))
               (T1 (SUM CARRY2)  HALF-ADDER (SUM1 C))
               (T2 (CARRY)       B-OR (CARRY1 CARRY2)))
              NIL)
  (HALF-ADDER (A B) (SUM CARRY)
              ((G0 (SUM)   B-XOR (A B))
               (G1 (CARRY) B-AND (A B)))
              NIL))
```

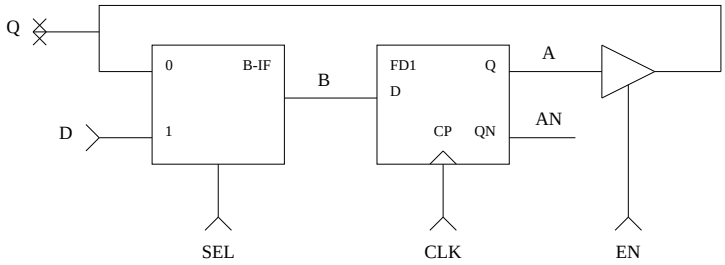
Notice that the definition of `HALF-ADDER` must occur after that of `FULL-ADDER` in the netlist, since `FULL-ADDER` references `HALF-ADDER`. The occurrences of `B-OR`, `B-XOR`, and `B-AND` need not be represented by module definitions in the netlist since these are hardware primitives defined by the HDL.

Whenever a module is introduced we define a predicate to recognize an appropriate defining netlist for that module. The recognizer is typically named with the module name, suffixed with the character "`&`". For example, the function `FULL-ADDER&`, shown below, is the recognizer for netlists defining the `FULL-ADDER` module.

```
(FULL-ADDER& NETLIST)
   =
(AND (EQUAL (LOOKUP-MODULE 'FULL-ADDER NETLIST) (FULL-ADDER*))
     (HALF-ADDER& (DELETE-MODULE 'FULL-ADDER NETLIST))
     (B-OR& (DELETE-MODULE 'FULL-ADDER NETLIST))))
```

`FULL-ADDER&` checks that `FULL-ADDER` has the previously given definition in the netlist and that the remainder of the netlist is appropriate for the component modules. The definition uses previously defined netlist recognizers for `HALF-ADDER` and for `B-OR`.

M1



M2

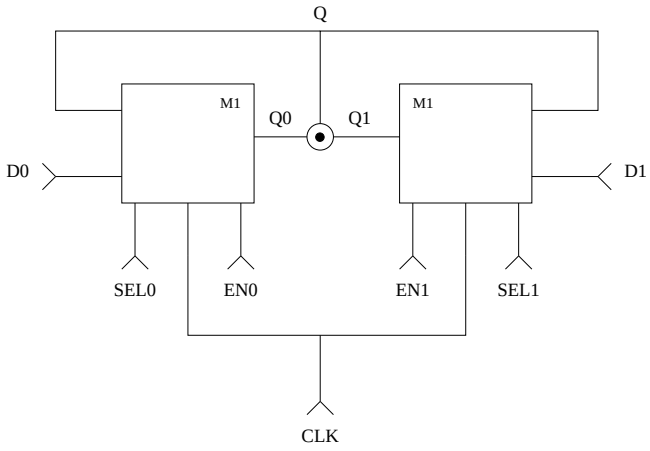


Figure 4: One-bit Latch with Tri-state Buffer Circuit and Bi-directional Bus Circuit

In addition to the purely combinational primitives such as `B-OR` and `B-AND`, our HDL contains primitive state-holding devices such as latches. Circuit modules that contain occurrences referencing either primitive state-holding devices or defined modules containing state-holding devices, must indicate which occurrences reference state-holding devices. These are listed in the fifth component of the circuit module. During evaluation of a circuit module we associate state, specified by occurrence names, with particular state-holding devices.

Consider the schematic circuits in Figure 4. The one-bit latch circuit `M1` is composed of three primitives: the `B-IF` selector primitive, the `FD1` one-bit latch primitive, and a tri-state buffer with an enable input. Our HDL representation of `M1` is as follows.

```
(M1*)
   =
'(M1 (CLK EN SEL D Q) (Q)
     ((MUX   (B)    B-IF  (SEL D Q))
      (LATCH (A AN) FD1   (B CLK))
      (TBUF  (Q)    T-BUF (EN A)))
     LATCH)
```

Notice that the last element in the circuit module is `LATCH`; this indicates that `LATCH` is the label of the (single) occurrence of a state-holding device `FD1` within this module. For modules that contain two or more occurrences of state-holding devices, these occurrences are represented in a list. All such occurrences must be listed.

Circuit module `M2`, also shown schematically in Figure 4, contains two occurrences of the (state-holding) `M1` circuit module defined above, and one occurrence of the wiring primitive `T-WIRE`. The occurrences that refer to circuit module `M1` are listed in the last argument of module `M2`. These represent the collected "state" of the module.

```
(M2*)
   =
'(M2 (CLK EN0 EN1 SEL0 SEL1 D0 D1) (Q)
     ((OCC0 (Q0) M1     (CLK EN0 SEL0 D0 Q))
      (OCC1 (Q1) M1     (CLK EN1 SEL1 D1 Q))
      (WIRE (Q)  T-WIRE (Q0 Q1)))
     (OCC0 OCC1))
```

A well-formed netlist contains a complete hierarchical description of a circuit in terms of primitive and defined modules. For example, an admissible netlist describing the complete full adder circuit is given below.

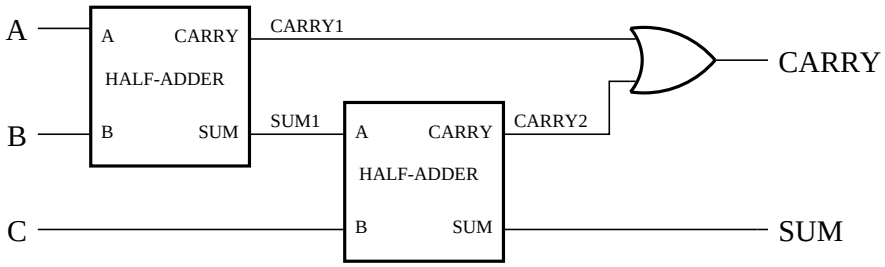


Figure 3: Full-Adder Circuit

The module `HALF-ADDER` is a structural description of the circuit with two inputs, `A` and `B`, and two outputs, `SUM` and `CARRY`. The module body is a list of module occurrences. Each occurrence consists of four elements: an occurrence name, a list of outputs, a defined circuit module name or primitive reference, and a list of inputs. In `HALF-ADDER` the first circuit module body occurrence is `(G0 (SUM) B-XOR (A B))`, which specifies that the output of the (primitive) reference `B-XOR (A B)` is connected to wire `SUM`. `G0` is the occurrence name. This `HALF-ADDER` circuit module does not contain any occurrences of state-holding devices; thus, the final component of the circuit module is `NIL`.

`HALF-ADDER` contains only primitive references; we can also compose defined modules. A schematic for a full-adder is presented in Figure 3. The `FULL-ADDER` circuit module contains two occurrences of the `HALF-ADDER` circuit module, meaning that two copies of `HALF-ADDER` are required to build the full-adder circuit. Our `FULL-ADDER` circuit specification is shown below. The internal names (wires) `SUM1`, `CARRY1`, and `CARRY2` interconnect the half-adders and the primitive `B-OR` gate.

```
(FULL-ADDER*)
    =
'(FULL-ADDER (A B C)
             (SUM CARRY)
             ((T0 (SUM1 CARRY1) HALF-ADDER (A B))
              (T1 (SUM CARRY2)  HALF-ADDER (SUM1 C))
              (T2 (CARRY)       B-OR (CARRY1 CARRY2)))
             NIL)
```

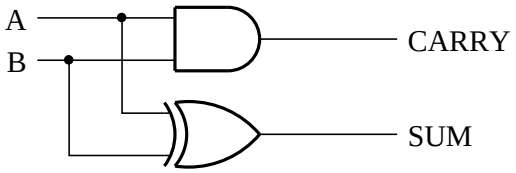


Figure 2: Half-Adder Circuit

As an example of a module within our HDL, below is a circuit constant for the half-adder whose schematic diagram is pictured in Figure 2. The circuit description is actually introduced as the body of a zero-ary function `HALF-ADDER*`. We follow a similar naming convention for all of our circuit modules. This allows us to refer to this constant in subsequent expressions as `(HALF-ADDER*)`.

```
(HALF-ADDER*)
    =
'(HALF-ADDER (A B)                          ; name, inputs
             (SUM CARRY)                    ; outputs
             ((G0 (SUM)   B-XOR (A B))      ; occurrences
              (G1 (CARRY) B-AND (A B)))
             NIL)                           ; state
```

the inputs and state. Syntactic restrictions on the language assure that this computation can be carried out efficiently and with a deterministic result.

The structuring concepts of the language specify how to combine primitive and defined modules into increasingly more complex designs. The resulting structures can be viewed as "wiring diagrams" accurately reflecting the hierarchical design structure of the resulting hardware. Such a modular design methodology is a crucial tool in managing the complexity of VLSI design. However, we take this one step further by providing a proof system that permits hierarchical verification of hardware designs. Thus, the advantages of modular design are inherited by the proof domain. These concepts are elaborated in subsequent sections.

4 Representing Circuits in our HDL

Circuit designs in our HDL are represented as constants within the Boyer-Moore logic. Functions written in the logic allow us to manipulate these constants—checking for syntactic admissibility, providing an operational interpretation of these constants as circuit designs, and enforcing various design constraints (such as fanout limitations). In this section we describe the representation of circuits.

We represent a circuit within our HDL as a netlist—a list of circuit modules. A well-formed circuit module consists of five elements:[3]

- a module name,
- a list of input names,
- a list of output names,
- a circuit module body in the form of a list of component module occurrences; and
- an occurrence name or a list of occurrence names.

Empty lists are represented as `NIL`.

All input and output names within a circuit module must be distinct, with the exception that a name may appear in both the input and output name lists, indicating a bi-directional wire. Component circuit modules may be primitive hardware devices defined by the HDL or previously defined user-supplied modules. The circuit module body is a list of occurrences—a set of wiring instructions connecting component circuit modules. No module may be self-referential. Primitives of the HDL include simple Boolean gates, registers, register files, and integrated circuit I/O buffers. A complete list of primitives is given in Section 10.

[3] An optional sixth argument is an association list of annotations. This argument is currently ignored by the evaluation software and will not be discussed further.

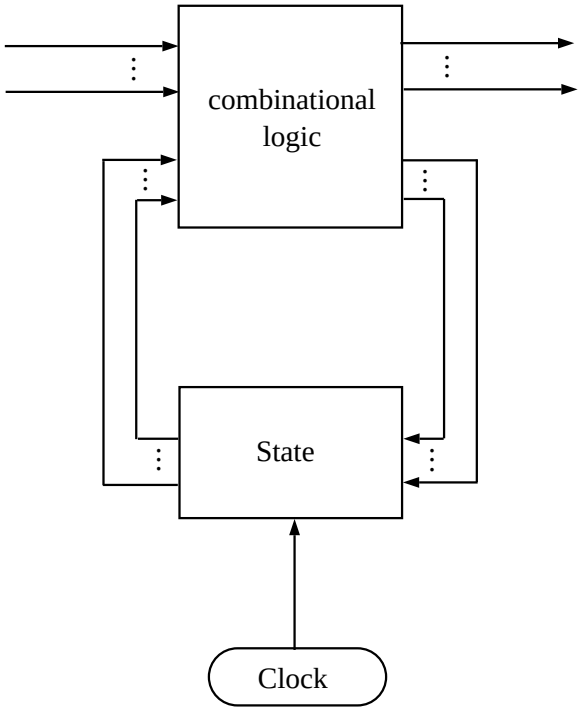


Figure 1: State Machine Model of a Hardware Module

expanding field.

There has been considerable work into formally modeling hardware description languages with an eye toward verified designs. Some languages studied include Zeus[19], SMAX[17], SML[9], CSML[13], ELLA[6] and VHDL[5, 18, 42]. Several of these have interfaced with commercial CAD tools. For example, SML has a postprocessor producing output compatible with the Berkeley VLSI design tools[10]. The VHDL work capitalizes on the growing availability of VHDL support tools.

The use of general-purpose mechanical theorem provers is widely recognized as useful in managing the proofs of increasingly complex VLSI designs. HOL[21] and the Boyer-Moore prover[2, 33] seem to be the most widely used in current hardware verification work, though other tools such as SDVS[35] and Clio[39], have also been used to verify substantial designs. In addition, there is considerable effort aimed at building special-purpose proof aids for reasoning about hardware circuits[12, 16].

3 Modeling Sequential Logic

Figure 1 shows a conventional view of a sequential logic circuit.[1] Along with the purely combinational circuit components, there are various *state-holding* components. The outputs of the circuit at any time are a function of the inputs and the current values of the state-holding components. The circuit is clocked, i.e., the outputs are also a function of time. The state-holding components take on new values only in response to a clock "tick."

Our HDL is an attempt to model this basic paradigm while abstracting away some of the complexity inherent in allowing the clock as an explicit signal in the model. We retain the combinational and sequential aspects of this model, but make the clock implicit. Thus, a module containing state-holding components is a finite state machine, while one without is purely combinational. Our HDL is a compromise between expressive power and "provability" that allows us to model a variety of hardware devices and prove interesting properties of our models.

Circuit descriptions in our HDL are built as a collection of *modules*. Each module is a specific instantiation of the basic circuit model depicted in Figure 1, where the clock is implicit.[2] Internally, each module comprises a collection of *occurrences* of primitives and other defined modules. In turn, a defined module can be referenced as a component of other defined modules. The primitives of the language include both combinational logic devices (eg. `NOR` gates) and state-holding devices (eg. flip-flops). Our evaluation paradigm is one of unit-clock simulation. On each "tick" of the clock (cycle of the simulation) we compute the new values of the outputs and state-holding components as a function of

[1] This diagram is adapted from [34].

[2] The clock signal is syntactically required in our circuit module descriptions. However, it is ignored by the evaluator.

The formalization of our HDL was motivated by the desire for the expressive power normally found in commercial hardware description languages, i.e., the ability to identify both active circuit elements and the interconnecting wiring networks. In previous work [22, 23, 24], we modeled sequential logic with Boolean-valued Boyer-Moore logic expressions. The intended hardware was derived from the structure of these expressions. State-holding modules could not be explicitly specified. Furthermore, fanout and other important engineering considerations could not be addressed. An alternative approach—using predicates to describe circuits—allows arbitrary circuit descriptions, but at the cost of a simulation capability.

In contrast, our HDL allows us to express circuits as logical *constants*, enabling us to provide mechanisms similar to those provided by CAD tools: simulation; synthesis; analysis of loadings, fanouts, and drive strengths; and syntax checking. Our HDL is our lowest-level model; that is, with our approach the most concrete means of describing a circuit is to represent it as a valid HDL constant. We can then manipulate and reason about these constants with mechanical tools built as functions in the Boyer-Moore logic. We can also mechanically translate these constants to other hardware description languages for physical implementation purposes.

We have used our HDL to express the implementation of the FM9001 microprocessor[25]. This implementation specification contains all internal gates, wires, test logic, and I/O circuits. For the FM9001 fabrication, we have mechanically translated our HDL-based implementation into a commercial CAD language and provided test vectors.

In the following section, we point to some related work. In Sections 3 and 4 we describe our overall model of circuits and their representation within our HDL. In Section 5 we describe informally the hardware interpreters that provide the semantics of our HDL. Sections 6 and 7 illustrate the specification and proof of properties of hardware designs. We then consider the design, specification and proof of circuit generator functions in Sections 8 and 9. Section 10 describes some limitations on our formalism and delimits the class of circuits that can be represented within our HDL. Section 11 describes the translation of our HDL circuit descriptions into some commercially available hardware description languages. Finally, we give some conclusions and observations on the value of this work in Section 12. The Boyer-Moore logic and theorem prover, along with some of the notation and function definitions used in the paper, are described in the appendix.

2 Related Work

A large number of researchers are working in the field of hardware verification and formal modeling of hardware. See Yoeli's recent tutorial[44] for an overview of the field. We mention here only a few of the threads of research from a rapidly

Abstract
We describe a hierarchical, occurrence-oriented hardware description language embedded within the Boyer-Moore logic. Within this formalism, we represent combinational and sequential circuits as list constants. An interpreter has been defined that gives meaning to circuit constants recognized by a well-formedness circuit predicate. Circuits can be verified with respect to their interpretations and we can write programs that manipulate HDL expressions. Instead of attempting to verify each circuit constant of interest, we often verify functions that synthesize circuit constants.

1 Introduction

The use of mathematical logic for modeling and reasoning about hardware designs promises assurance of circuit correctness beyond what is available from current state-of-practice techniques. The development and use of formal techniques in hardware design is spreading [4, 8, 12, 14, 15, 26, 27, 29, 38]. This approach to circuit validation is known generally as *hardware verification*. Circuits with the complexity of microprocessors[4, 22, 30, 39] have been given mathematical specifications, and their designs have been proved to implement their specifications. Yet, the transfer of hardware verification techniques to commercial engineering practice has been hampered by such factors as the use of non-standard notations, inaccessibility of the tools, and the significant mathematical sophistication required to use these approaches. In addition, formal techniques have been directed at only selected aspects of the design process. Important hardware characteristics such as testability and I/O behavior have been largely neglected by the formal hardware modeling and verification community.

We have attempted to address some of these issues by formalizing a subset of a conventional CAD-like sequential hardware description language (HDL). We provide a formal circuit syntax, a formal semantics, and a means of translating circuit descriptions to a commercial CAD language. Our HDL definition encompasses the notions of circuit behavior, delay, fanout, logical values, loading, modularity, and circuit module hierarchy.

Our HDL is formally embedded within the Boyer-Moore logic[1, 2]. Consequently, we have available a formalism for specifying correctness properties of our circuits and a theorem prover to support proofs about our circuit designs. Circuits are represented as constants within the logic, which is also used to define the syntax and semantics of our circuit constants. We employ the Boyer-Moore theorem prover to mechanically manage our database of specifications and to check our proofs. We are also able to construct and verify functions that generate correct circuit descriptions in our HDL. For instance, we have proved the correctness of a function that produces ALU circuits for arbitrary word sizes[7]. We know in advance that any circuit constructed by this ALU-producing function is correct.

An Introduction to a Formally Defined Hardware Description Language

Bishop C. Brock, Warren A. Hunt, Jr., William D. Young

Technical Report 76 April, 1992

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

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

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

This paper appears in the proceedings of the International Conference in Theorem Provers in Circuit Design, held in Nijmegen, the Netherlands, 22-24 June, 1992. This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406. 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.