[11] Steven D. Johnson. Manipulating logical organization with system factorizations. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects.*, volume 408 of *Lecture Notes in Computer Science*, pages 259–280. Springer Verlag, 1989.

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

[13] Warren A. Hunt Jr. *FM8501: A Verified Microprocessor.* PhD thesis, University of Texas at Austin, December 1985.

[14] Warren A. Hunt Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, December 1989.

[15] 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.*, Lecture Notes in Computer Science, pages 281–305. Springer Verlag, 1989.

[16] Matt Kaufmann. A hardware reset lemma and its proof. Internal Note 230, Computational Logic, Inc., May 1991.

[17] J S. Moore. Mechanically verified hardware implementing an 8-bit parallel io byzantine agreement processor. Technical Report NASA CR-189588, NASA, 1992.

[18] Mary Sheeran. μ-FP–An algebraic VLSI design language. Technical Report PRG-39, Oxford University Computing Laboratory, September 1984.

References

[1] 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.

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

[3] 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.*, volume 408 of *Lecture Notes in Computer Science*, pages 128–150. Springer Verlag, 1989.

[4] R.E. Bryant. Verification of synchronous circuits by symbolic logic simulation. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects.*, volume 408 of *Lecture Notes in Computer Science*, pages 14–24. Springer Verlag, 1989.

[5] Avra Cohn. A proof of correctness of the viper microprocessor: The first level. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71. Kluwer Academic Publishers, Boston, MA, 1988.

[6] W.J. Cullyer. Implementing safety critical systems: The viper microprocessor. Divisional Memo (CC2) 411-87, Royal Signals and Radar Establishment, Malvern, Worcestershire (United Kingdom), August 1987.

[7] 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.

[8] Donald I. Good. Mathematical forecasting. In Christine Anderson and Merlin Dorfman, editors, *Aerospace Software Engineering*. American Institute of Aeronautics and Astronautics, Inc., 1991.

[9] M. Gordon. HOL: A proof generating system for higher-order logic. Technical Report 103, University of Cambridge, Computer Laboratory, 1987.

[10] 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.

possible to prove the monotonicity property we mentioned earlier. We believe that the proof of monotonicity property of the HDL is one check on the validity of our hardware model.

The weakness of our HDL forced us to restrict ourselves to implementing a device with a single clock. This restriction precludes the formalization of any multi-phased clocked device, e.g., a RISC microprocessor. In addition, our HDL does not admit the use of both rising-edge and falling-edge clocked registers – we consider every state-holding device to update its internal state simultaneously. The FM9001 implementation does include one level-sensitive memory element, the register file. We chose to include this non-standard primitive because a level-sensitive memory of the size of our register file can be manufactured in one half of the space required by 512 single-bit latches. We implemented a circuit around the register file so that we could use the register file just like all of the other state-holding device primitives.

We expect that if we were to design another microprocessor of similar complexity it would not take nearly as long as this effort. Much of our effort was spent in refining our HDL and making proofs about HDL circuits as automatic as possible. We spent a great deal of time "engineering" our HDL proof system so it would be simple for others to use. In fact, a circuit for Byzantine agreement [17] and a combination lock circuit [16] have been specified with our HDL and these circuits were verified by Boyer-Moore theorem-prover users other than ourselves.

We have recently formalized single stuck-at faults for the combinational portion of our HDL. We believe that this is an area that should be integrated with the design and verification process. We consider this potential integration as another thread that should be sewn into the formula manual cloth.

10 Acknowledgments

This work was done in collaboration with Bishop C. Brock until his departure in July of 1991. Bishop Brock deserves equal credit for this effort, and without his innovation and theorem proving ability this work would have not been possible. This paper was written entirely by Hunt. Brock is presently on an around-the-world bicycle trip, and thus was unable to review this note.

Matt Kaufmann was responsible for the formalization and proof of the monotonicity of our HDL. Ann Siebert built the well-formed HDL circuit recognizer. Fay Goytowski engineered the formalization of our stuck-at test model.

This work was sponsored 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.

the first two lines of the chart just below.

	Prettyprinted Lines
User-level semantics of FM9001	915
Statement of theorem	197
Semantics of HDL	3459
FM9001 implementation description	3479
Existential witness for the clock	1942
Total:	9992

To get a rough measure of the size of the FM9001 proof, J Moore modified the theorem prover to count its smallest step. By adopting a suitable collection of derived inference rules, a formal proof can be constructed with one line for each step. Among our one-step rules are instantiation, *modus ponens*, generalized equality substitution, tautology recognition, and cross-multiplication and addition of inequalities.

The sum of the proof steps in all the formal proofs done for FM9001 is 6,100,315. That is, more than six million lines of formal proof were "virtually constructed." It would be easy to increase this number by reducing the size of our proof steps. For example, the cost of substitution-of-equals-for-equals in HOL [9] is not constant (it is one in our count of proof steps) but is proportional to the depth at which the target occurrence is found. Similarly, we recognize very large `IF` expressions as tautologies but charge only one step. We have expressions containing more than 50 `IF` expressions through which there are more than 10^9 branches—but because the branches were pruned dynamically, the theorem prover did not have to explore them all.

The total number of primitive proof steps investigated by the theorem prover during its search for the FM9001 proof is 19,165,122. This means that 32% of all the investigated steps were actually "kept." The term clausifier was called 80,390 times. The function that determines the type of a term was called 14,699,506 times and the rewriter was called 11,624,455 times.

9 Conclusions

A netlist implementation of the FM9001 microprocessor has been mechanically checked by the Boyer-Moore theorem proving system to implement its formal specification. The more important result of this effort is the notion that the FM9001 specification and implementation represents our first *formula manual*. Although we consider the verification of circuit designs to be important, we believe the formalization of design requirements to be more valuable.

We know of no other work where the complete syntax and semantics of a HDL have been described. Without this formalization, it would not have been

An abstract representation of the top-level theorem specifying the correctness of an implementation for the FM9001 specification is below.

$$\exists implementation\ \exists clock$$
$$FM9001_specification(user_state, n)$$
$$=$$
$$map_up(simulate(map_down(user_state), implementation, clock))$$

The *implementation* for which we have proved this theorem is, of course, the one previously described. We remind the reader that there is a time abstraction between the user and two-valued levels. The *clock* argument above accommodates the different rates of the user and lower levels. The natural number n indicates the number of user-level instructions to be executed.

Our FM9001 proof script contains 2957 entries that expand into 4851 theorem prover events. The total time required for the Boyer-Moore theorem prover to check the proof of FM9001's implementation with respect to its specification is about 24 hours on a Sun Microsystems 3/60.

The following chart is a partial reproduction of a chart in Appendix III of *A Computational Logic Handbook*, by Boyer and Moore [2]. We added the last line. This information was constructed by a static analysis of the final FM9001 library produced by the Boyer-Moore theorem prover.

```
                  Number of lines in understandable statement
                  |     Concept depth of statement
                  |     |     Max concept depth in proof
                  |     |     |     Number of supporters
                  |     |     |     |     Lines of supporters
                  |     |     |     |      |    Depth of proof
                  |     |     |     |      |    |
85 FM8501   991   157   152   230   2171   18
86 Goedel   864    48 40414  1741  20002   58
91 FM9001  1112   120   128  1894  28784   46
```

The "understandable statement" of a theorem is the prettyprinted text of all the definitions used in the theorem except for Boyer-Moore primitives. The number of lines for the understandable statement of the FM9001 proof could be considered to be as high as 9992. The 1112 lines reported above is the sum of

automatically from a description of what internal resources are to be used each clock cycle. However, in this case we also generate a list of lemmas that must be established to ensure the correctness of the control logic. The lemmas are mechanically proved without manual intervention.

The use of synthesizers wherever possible has made most of the FM9001 pieces reusable. Only a few pieces are fixed in their word size. To implement another similarly complex microprocessor would require much less effort than we spent implementing the FM9001 because of the existence of many verified synthesizers.

The FM9001 implementation netlist is composed of 85 modules that collectively reference 1800 primitives of 48 different types. On average, the output of a primitive is connected to 3.4 other primitives. The FM9001 has 95 I/O pins, of which 32 are bi-directional. It is this netlist that we have mechanically checked to implement the FM9001 specification.

To allow LSI Logic to build the FM9001, we translate this netlist into NDL as discussed earlier. The resulting NDL specification of the FM9001 contains over 91000 characters in 2215 lines. This NDL specification is used by LSI Logic as a wiring guide for an actual implementation. In addition to the NDL-based specification, we provide test vectors. Even a logically correct implementation must be tested to ensure that there were no flaws introduced during the manufacturing process.

To facilitate testing, the 248 internal single-bit latches are connected in a scan chain. Through the use of an external test control input, it is possible to load or read every bit in the chain. The contents of the internal 16 word by 32-bit register file can be loaded and retrieved by using the scan chain along with two, external, dedicated, register file control signals . That is, it is possible to stop the FM9001 implementation on any clock cycle, and read and/or alter any or all of the internal state.

8 The FM9001 Mechanical Proof

The proof of the FM9001 microprocessor was mechanically checked by the Boyer-Moore theorem prover. The proof is large by mechanical standards and impossible to do by hand. We present a top-level statement of the correctness of the FM9001 implementation and describe the size of the mechanical proof.

are false and undefined, then the result is false. If we consider the primitive *exclusive-or* gate, `DUAL-EVAL` produces an undefined result if either input argument is undefined.

We have proved that `DUAL-EVAL` is monotonic with respect to an ordering in which the undefined value is smaller than everything else. It was necessary to first prove that every primitive was monotonic and then to prove that well-formed netlists were also monotonic. Using this property and the ability of some primitive gates to prevent the propagation of undefined values, we were able to prove that the FM9001 microprocessor could be reset, even if the initial state was undefined. We prove that the FM9001 can be reset by simulating the FM9001 netlist with `DUAL-EVAL` using a completely undefined initial state.

By referring back to Figure 1, we can now describe the difference between the two-valued and the four-valued levels. The four-valued level is just a functional abstraction of our netlist level – this level implements a four-valued logic – and the proof that the netlist level implements the four-valued level is straightforward. After the FM9001 has been reset, every state-holding device has been proved to contain only Boolean values; we assume that the memory contains only Boolean values. The two-valued level admits only Boolean values. To prove that we can lift ourselves from the four-valued level to the two-valued level, we must prove that we have a completely Boolean state and that the memory interface only supplies Boolean values (during memory read operations.) It is between these two levels that we must prove that the memory interface protocol, implemented by the FM9001, is correct for our memory model; otherwise, we are not able to prove that the memory always provides Boolean data.

7 The FM9001 Circuit Netlist

The implementation of the FM9001 is described as a netlist containing a reference for every primitive gate, latch, register, I/O buffer, and wire that is required for a FM9001 implementation. Both the syntax and the semantics of this netlist have been checked: the netlist is well-formed and it has been mechanically checked to implement the FM9001 specification.

The FM9001 netlist is actually constructed by executing a function that generates the FM9001 netlist. Much of the netlist is synthesized. The ALU, the largest internal block of combinational logic, is defined by a generator parameterized by the word size and the structure of the propagate-generate look-ahead logic. We have mechanically proved that our ALU synthesizer constructs correct ALU's for all word sizes and look-ahead structures. Registers, selectors, I/O pads, and other regular structures are also synthesized. Instead of proving the correctness of the circuits produced by the synthesizer programs, we prove the correctness of the synthesizer programs themselves. In this way we know that the resulting circuits are correct and we do not have to inspect the resulting circuit modules. We also synthesize our control logic; it is produced

output names, translating the primitive reference names, and adding keywords for NDL. 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.

```
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))
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;

END COMPILE;
END;
```

6 The Unit-clock Simulator

The function that implements our unit-clock simulator is named `DUAL-EVAL`. This is because `DUAL-EVAL` must evaluate a netlist twice to compute new values for the state-holding devices. A module's inputs may not be available during the first evaluation pass because their values may not have been computed; that is, their computation occurs later in the evaluation of the netlist. At the start of the second pass, all outputs of every module will have been computed and are available as inputs to any other module.

`DUAL-EVAL` implements a four-valued simulator; it recognizes four distinct logical values: true, false, floating, and undefined. Undefined values given as arguments to primitives are passed on by `DUAL-EVAL` as appropriate. For instance, if a two-input Boolean *and* gate is evaluated with one input being undefined and the other true, then an undefined result is returned; however, if the inputs

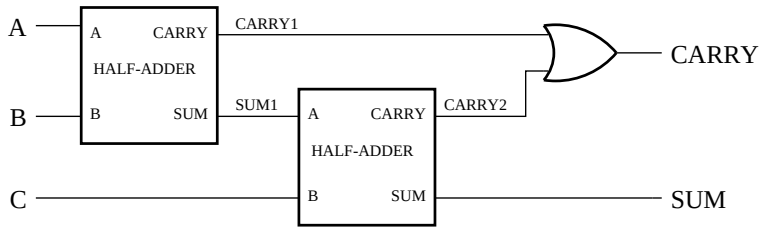


Figure 4: Full-Adder Circuit

```
'(HALF-ADDER (A B)
             (SUM CARRY)
             (( G0(SUM)   B-XOR(A B))
              ( G1(CARRY) B-AND(A B)))
             NIL)
```

The `HALF-ADDER` module has two inputs named `A` and `B` and two outputs named `SUM` and `CARRY`. The circuit module body is a list of two occurrences. The first circuit module body occurrence is `( G0(SUM) B-XOR(A B))`; the output of the primitive reference `B-XOR(A B)` is `SUM`; `G0` is the occurrence name. The final `NIL` in the `HALF-ADDER` module indicates that this module contains no state-holding devices.

A full-adder schematic is shown in Figure 4. `FULL-ADDER` references the `HALF-ADDER` circuit module twice.

```
'(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)
```

We introduce the internal names (wires) `SUM1`, `CARRY1`, and `CARRY2` to interconnect the half-adders and the primitive `B-OR` gate. In addition to the purely combinational primitives such as `B-OR`, our HDL contains primitive state-holding devices such as latches. We also have primitives for tri-state busses, I/O buffers, scanned latches, and clock buffers.

Below is a an example of a translation of our format to LSI Logic, Inc., Network Description Language (NDL). A netlist containing `FULL-ADDER` and `HALF-ADDER` is positioned on the right and the result of our translation is on the left. To have the FM9001 fabricated, we have written a translator in Common Lisp to translate the entire FM9001 implementation netlist into NDL. The translation is primarily concerned with added commas to separate input and

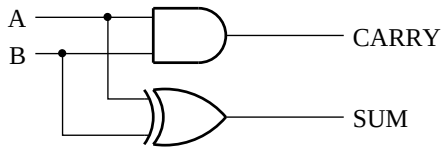


Figure 3: Half-Adder Circuit

provide a structure for our circuit descriptions. If we replace `B` in the previous list by `'(D E F)` we get `'(A (D E F) C)`.

A well-formed netlist is composed of a list of well-formed modules. Every module name in the netlist must be unique and different from every primitive name. Each module is composed of five elements: a module name, a module input list, a module output list, a body containing a list of occurrences, and either a reference to a single state-holding occurrence or a list of state-holding occurrence names. Module input and output names must be distinct; however, a name may appear in both the input and output name list, indicating a bi-directional name. A circuit module body is a list of occurrences that reference other modules or primitives. The occurrence list describes how these are interconnected or "wired" together. Each occurrence in a module body contains four fields: an occurrence name, an output list, a module reference, and an input list. All occurrence names in a module body must be distinct. A module reference in every occurrence must refer to either a primitive or another module defined later in the netlist. Our HDL definition includes simple Boolean gates, registers, register files, and integrated circuit I/O buffers as primitives. Output names of an occurrence must be distinct from any module input name (except for bi-directional nets), but they can include module output names. No name may appear twice in occurrence output lists; that is, a name may not be set by more than one occurrence output. The input list of an occurrence may reference only the module input names or other wire names as set by other occurrence outputs. The last element of a module definition must have an entry for every occurrence containing state-holding devices. If only one occurrence contains state, then that occurrence name is the module's fifth element; otherwise, the state-holding occurrence names are combined into a list.

Below we present our HDL description of a full adder composed of two half adders. Below is our formal module definition of the half-adder schematic in Figure 3.

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 [12] 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.

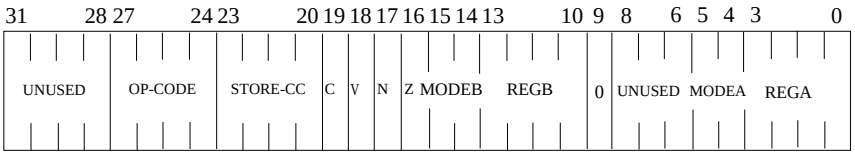
While the Boyer-Moore logic is formal, the theorem-prover does not construct a formal proof. To the best of their abilities, Boyer and Moore have convinced themselves by the traditional, rigorous methods of informal mathematics of the following claim: "If the theorem prover asserts it has proved a formula, then there exists a formal proof of that formula." The validity of their claim has been subjected to the scrutiny of the "social process" of the mathematics and computer science communities through books and classes, and the full distribution of the theorem prover code, for almost two decades without serious challenge. This time span is important, as this is not a long time for certification in mathematics. It must be noted that the Boyer-Moore theorem prover is about 15,000 lines of Common Lisp, and it is quite possible that the proving mechanism in the theorem prover has soundness flaws.

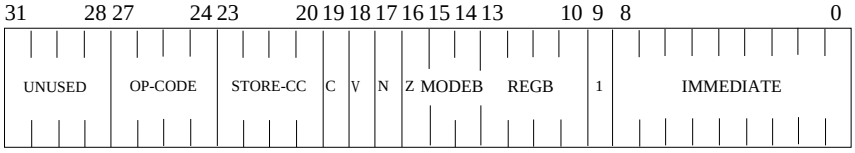
We use the theorem prover as a proof checker. We lead it to difficult theorems by providing it with a graduated sequence of more and more difficult lemmas until a final result can be established.

5 Circuit Netlists

We use our formally defined HDL to define the FM9001 implementation. Our definition of the HDL contains two main pieces: a recognizer of well-formed circuits and the unit-clock simulator. The well-formed circuit recognizer predicate checks a netlist for a number of static circuit properties. The simulator, when given a netlist, a module reference, the module's input values, and the module internal state, computes both the outputs and the next internal state. The unit-clock simulator provides a circuit simulation capability.

HDL circuits are written using the Lisp quote notation. For example, `'(A B C)` is a list of three literal atoms; its second element is `B`. We use nested lists to





MODE	OPERAND	DESCRIPTION
00	Rn	Register Direct
01	(Rn)	Register Indirect
10	-(Rn)	Register Indirect Pre-decrement
11	(Rn)+	Register Indirect Post-increment

OP-CODE	OPERATION	DESCRIPTION	STORE-CC	CONDITION
0000	b <- a	Move	0000	Carry clear
0001	b <- a + 1	Increment	0001	Carry set
0010	b <- a + b + c	Add with carry	0010	Overflow clear
0011	b <- b + a	Add	0011	Overflow set
0100	b <- 0 - a	Negation	0100	Not negative
0101	b <- a - 1	Decrement	0101	Negative
0110	b <- b - a - c	Subtract with borrow	0110	Not zero
0111	b <- b - a	Subtract	0111	Zero
1000	b <- a >> 1	Rotate right through carry	1000	Higher
1001	b <- a >> 1	Arithmetic shift right	1001	Lower or same
1010	b <- a >> 1	Logical shift right	1010	Greater or equal
1011	b <- b XOR a	XOR	1011	Less
1100	b <- b OR a	OR	1100	Greater
1101	b <- b AND a	AND	1101	Less or equal
1110	b <- NOT a	NOT	1110	True
1111	b <- a	Move	1111	False

Figure 2: Instruction Word Format

increment, and register indirect with post-increment. Thus, the FM9001 has five addressing modes for operand A and four addressing modes for operand B. Every addressing mode works with all instructions. There are 15 different arithmetic instructions. Each instruction contains four bits that allow the arithmetic flags, carry, overflow, zero, and negative, to be selectively updated. Also, each instruction contains four bits that specify whether the result computed by an instruction is stored, based on the values of the arithmetic flags at the beginning of an instruction.

The user-visible state of the FM9001 is shown as a part of the user level in Figure 1. This state includes a general-purpose 16-element register file, four arithmetic flags, and the external memory. Register 15 is overloaded to contain the program counter address, but in all other respects operates like the other registers. To perform a conditional branch, operand A is added to the program counter, and the result is conditionally stored based on the `STORE-CC` field of the instruction. Since the results of any instruction may be conditionally stored, there are no differences between control and data operations. This is even true of the program counter because the implementation allows any of the 16 registers to be used as the program counter.

The FM9001 specification is derived from the FM8502 specification [14]. There are three main differences. The FM9001 has the ability to conditionally store any result – only conditional moves were permitted in the FM8502. The FM9001 interleaves the post-increment operations; the FM8502 post-increment operations were done after both operands were fetched. And finally, the FM9001 has an immediate data mode for operand A; the FM8502 did not have this feature. The ALU operations remained the same as did the fetch-execute cycle.

The two-valued and the four-valued specification levels are artifacts of the approach we used to mechanically check the implementation of the FM9001. The two-valued level can be thought of as a Boolean model of the implementation with the tri-state memory interface bus abstracted away. The four-valued level has functionality similar to the two-valued level, except the tri-state memory bus is included. It is at this level that we prove that the FM9001 can be reset; we discuss this later. The netlist specification level describes the actual FM9001 implementation in terms of Boolean gates and latches, I/O buffers, and test logic. We translate our netlist into vendor-specific languages so a manufacturer can build a physical implementation.

4 The Boyer-Moore Logic

We express our netlists as Boyer-Moore list constants. Before we explain our netlist syntax, we introduce the Boyer-Moore logic. The Boyer-Moore logic 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, provided they terminate. The logic includes several built-in data

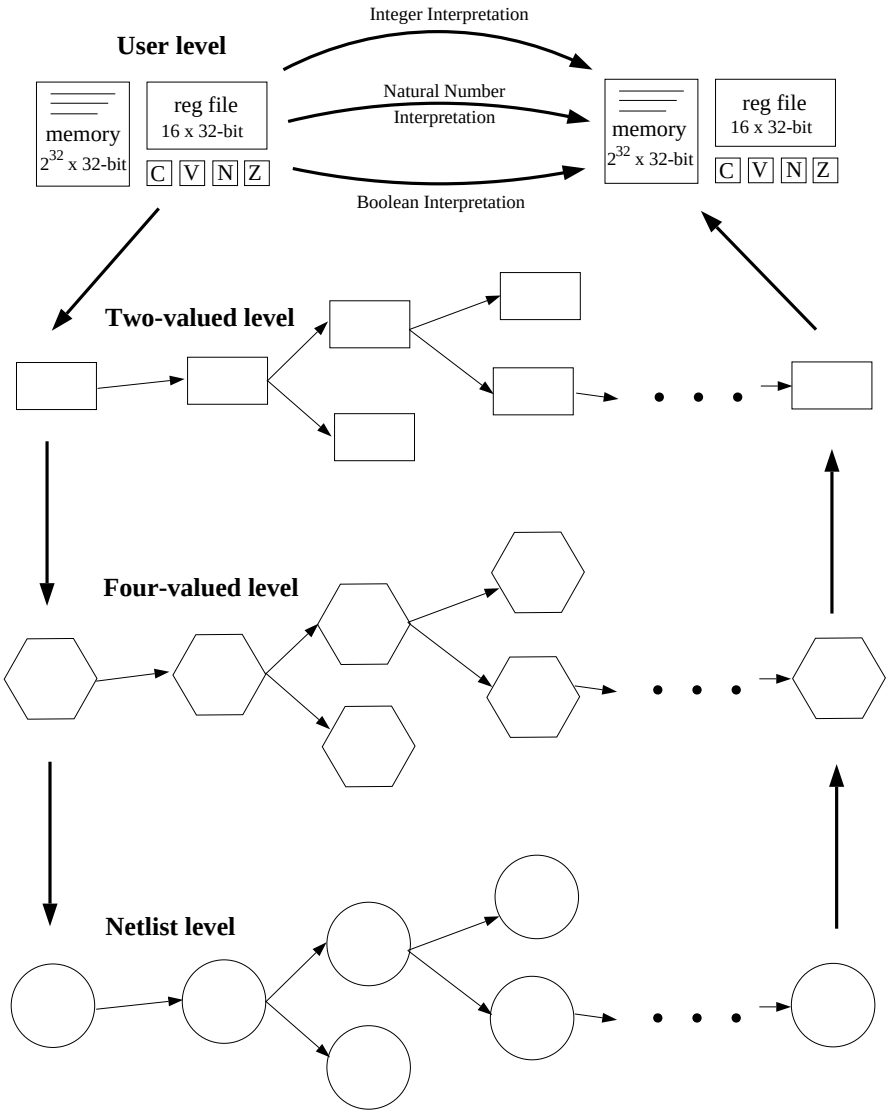


Figure 1: Specification Levels

model of circuit implementations, well-formed circuits cannot be recognized, nor can the model be thoroughly studied for its mathematical consistency. Clearly, our HDL is a model of physical reality, and as such, it should be subjected to thorough study; our HDL should be inspected by digital design engineers to determine whether our model corresponds with physical reality. Our model represents circuits as interconnected networks of simple Boolean connectives and storage devices. Circuits at this simple Boolean level are still far removed from actual physical implementation.

Our HDL has been designed with a structure similar to that of commercial HDLs, thus allowing a simple mechanical means of converting our design specifications into existing informally defined HDLs. Clearly this conversion process has no formal basis; therefore, it is important that conversions can be performed with a simple, mechanical, believable process.

3 The FM9001 Microprocessor Specification

The FM9001 is specified at four levels of abstraction – user, two-valued, four-valued, and netlist – shown in Figure 1. There are three user-level specifications: Boolean, natural number, and integer. Each gives identical results, but by way of different interpretations of each FM9001 user-level operation. All of the horizontal arrows represent interpreters that repeatedly take a state and map it to a new state. The netlist level state, represented with circles, is the actual state in the FM9001 implementation. The two-valued and four-valued levels have a slightly more abstract representation of the netlist state. The user-level state contains only what is available to a FM9001 programmer. The user-level specifications are interpreters that step once per instruction and provide the programmer's view of the FM9001. The two-valued and four-valued levels are different abstractions of the netlist level; these interpreters step once per machine clock cycle. The netlist itself is not an interpreter, but it can be evaluated on a clock-by-clock basis with our netlist simulator. The vertical arrows represent trivial mapping functions except between the user and two-valued levels. Mapping here converts between the user visible state and the state in the FM9001 implementation; additionally, there is a time abstraction because of the different step rates between these two levels.

Informally, the FM9001 is a simple 32-bit microprocessor with a two address architecture. Each FM9001 instruction fetches operand A, and operand B if necessary, computes a result, and stores the result in the location referenced by operand B. Figure 2 shows the two FM9001 instruction formats, a two-address instruction or an immediate datum instruction. The formats only differ in selection of operand A. In the immediate datum case there is a nine-bit datum that is sign-extended to 32-bits; otherwise, operand A is selected in the same manner as operand B. Except for the immediate datum mode there are four addressing modes: register direct, register indirect, register indirect with pre-

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 mechanically translate these constants to other hardware description languages for physical implementation purposes.

The FM9001 microprocessor is a general-purpose, two-address, 32-bit microprocessor with five addressing modes. Our implementation of the FM9001 has 32 address lines, a 32-bit data bus, 32-bit internal registers, and Boolean arithmetic flags. The FM9001 user-level specification is an instruction interpreter that steps once per instruction. An implementation of the FM9001 has been specified as a well-formed circuit netlist. We have led the Boyer-Moore theorem-proving system to a mechanically checked proof that for all possible simulations of the FM9001 netlist, it implements its specification, given certain restrictions on the memory interface. To facilitate manufacturing we convert the FM9001 netlist into a commercial netlist language.

In this note we present an introduction to our HDL and the FM9001. We describe the approach of using an HDL like ours for the mechanical verification of hardware designs, and we use the FM9001 microprocessor verification effort to demonstrate this approach. We also document the size of this effort and conclude with some comments about our experience.

2 Hardware Verification

Our "hardware verification" approach advocates the use of formal logic for both hardware designs and their more abstract specifications. We envision providing a mathematical statement, which we call a *formula manual*, that completely specifies the operation of a hardware component. Much effort is being invested in pursuing this goal [1, 3, 4, 5, 6, 7, 10, 11, 18]; however, most of the effort is directed toward proving the logical correctness of various circuits. Conceptually, we think of a formula manual specifying much more than logical correctness. For instance, imagine a microprocessor user's manual containing a series of formulas that describe 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 formally checked to meet every specification contained in the formula manual. The formula manual would then provide a precise specification that would allow hardware engineers to use implementations with confidence and would permit software engineers to accurately predict the results of programming the device. The formalization of our HDL represents a very modest attempt toward formalizing the hardware design process.

Our HDL explicitly models (with varying degrees of precision) circuit fanout, loading, test circuitry, I/O ports, and circuit hierarchy [8]. Without a formal

Abstract

A synchronous, hierarchical, occurrence-oriented, hardware description language (HDL) has been formalized with the Boyer-Moore logic. Well-formed HDL circuits are recognized by a predicate, and a unit-clock simulator defines the meaning of circuits expressed in the HDL. This HDL has been used to specify an implementation of the FM9001 microprocessor that has been mechanically proved to implement the FM9001 instruction-level specification. All proofs were mechanically checked using the Boyer-Moore theorem-proving system. The formalization of the HDL, the FM9001 user-level specification, and the FM9001 HDL implementation architecture specification required more than seven hundred function definitions. The mechanical proof is composed of thousands of theorem prover proof requests and millions of theorem prover inference steps.

1 Introduction

The formalization of a hierarchical, occurrence-oriented, hardware description language has been specified using the Boyer-Moore logic [2]. Circuits are represented as Boyer-Moore list constants, and the Boyer-Moore logic is used to define circuit semantics. An implementation of the FM9001 microprocessor has been specified with this hardware description language. By using the Boyer-Moore theorem proving system, we have mechanically checked that our HDL-based FM9001 implementation specification is both an admissible circuit and that it implements its user-level specification.

Our hardware description language (HDL) only admits sequential circuits with a single clock; all state-holding devices update their internal states simultaneously. Well-formed circuits are recognized by a predicate that inspects circuits for well-formed names, the absence of combinational loops, loading and fanout violations, wire type mismatches, clock distribution, and other circuit parameters. The meaning of interconnected collections of well-formed circuits (netlists) is defined by a unit-clock simulator. Well-formed circuits may be mechanically analyzed using the Boyer-Moore theorem-proving system.

The formalization of our HDL was motivated by the desire for the expressive power normally found in hardware description languages, i.e., the ability to identify both active circuit elements and the interconnecting wiring networks. In our previous work [13, 14, 15] we modeled combinational logic with Boolean Boyer-Moore logic expressions. The intended hardware derived from the structure of these expressions. State-holding modules could not be described this way and all notion of state was confined to a "register-transfer level." Furthermore, fanout and other important engineering considerations could not be addressed. Another approach we investigated was the use of predicates – this allows arbitrary circuit descriptions, but then no direct simulation capability is available. By expressing circuits as logical constants, we are able to provide

A Formal HDL and its use in the FM9001 Verification

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

Technical Report 79 July, 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

This paper appeared in the Philosophical Transactions of the Royal Society of London. Series A, Vol. 339. This work was sponsored 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.