The DUAL-EVAL Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor

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

Computational Logic, Inc.
1717 W. 6th St., Suite 290
Austin, TX 78703-4776
U.S.A.
Tel: +1 512 322 9951
FAX: +1 512 322 0656
email: hunt@cli.com, brock@cli.com

Abstract — We present the full formal semantics of the DUAL-EVAL hardware description language. DUAL-EVAL is a hierarchical, occurrence-oriented simulator for synchronous Mealy machines. We briefly describe the FM9001 microprocessor, whose design has been formally specified with the DUAL-EVAL language and mechanically proved correct with respect to a behavioral specification. The FM9001 has been fabricated as a CMOS ASIC and tested extensively.

I. Introduction

The use of mathematical logic for modeling and reasoning about hardware designs promises assurance of circuit correctness beyond what is available from current techniques. The development and use of formal techniques in hardware design is spreading [3, 11, 12, 14, 15, 17, 20, 26, 30, 34, 36, 39]. This approach to circuit validation is known generally as *hardware verification*. Various hardware description languages have been formalized using a variety of formal methods [8, 5, 16, 27, 21]. Circuits with the complexity of microprocessors [3, 22, 25, 35, 36, 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.

Industry-standard techniques for assuring the quality of hardware are simulation and testing. These are crucial tools, but we hope to model hardware mathematically and prove the correctness of our hardware models. Our goal is to work toward the point where documentation for a hardware system is a *formula manual*, a collection of fully formal specifications that provide a precise and complete basis for using the device. To assure that the formula manual is correct, it will be necessary to *prove* that a design meets *all* of its specification. In fact, a test of the utility of a formula manual also includes the possibility of using it as the basis for constructing a hierarchy of formally verified systems, including larger hardware structures, operating systems, and user applications.

We have attempted to address some of these issues by undertaking the formal specification, verification, and physical implementation of the FM9001 microprocessor. The FM9001 is a general-purpose 32-bit microprocessor whose gate-level netlist design implementation was developed using a theorem-proving environment in conjunction with a traditional CAD system. The behavioral specification for the FM9001, the definition of the hardware description language (HDL) used to represent the design of the FM9001, the simulator for the HDL, and the verification of the FM9001 were all carried out using the Boyer-Moore theorem-proving system Nqthm [6]. The FM9001 netlist was mechanically translated to LSI Logic's Netlist Description Language and implemented by LSI Logic, Inc., as a CMOS gate-array. Rigorous testing has not uncovered any situation where the manufactured device fails to meet its specification. The FM9001 also serves as the target for the verified assembler, Piton [31], which in turn serves as the target of the verified μ-Gypsy compiler [41]. This document presents the details of the

DUAL-EVAL hardware description language and summarizes the FM9001 development, its specification, and its verification.

II. History of the Project

The FM9001 effort began in earnest in November 1989 when we defined a simulator, which we called DUAL-EVAL [9], for a language that was general enough to represent the design of a microprocessor along with its memory system. Representing circuits as data was a turning point in our efforts at hardware specification. Previously Hunt [22, 23] had represented hardware with functions in the Boyer-Moore logic. We had also investigated representing circuits as predicates [7], in the tradition of the HOL hardware verification community [13]. Although both of these approaches were successfully used in significant hardware verification examples, neither approach permits direct, formal operations on the circuit descriptions.

Immediately prior to the definition of the DUAL-EVAL simulator and language recognizer we had defined several different hardware description languages which differed in representational power. For example, the language/simulator HEVAL [8] was restricted to combinational logic circuits only. We also defined WAVE-EVAL, which worked similarly to an event-driven simulator. WAVE-EVAL was more general than DUAL-EVAL, but we considered its semantics too difficult to use given the size of our task. Once we had defined DUAL-EVAL, we developed techniques to simplify the specification and verification of circuits described in the DUAL-EVAL language, at least until we had automated the process sufficiently to verify the FM9001 design. In actuality, the evolving FM9001 design was a "technology driver" for the development of the DUAL-EVAL language.

Another part of the HDL development was the evolution of its language recognizer. At first, the recognizer checked for proper syntax and absence of combinational circuit loops; however, over time the recognizer was extended to check for loading and fanout violations, timing properties, restrictions on names, and a host of other properties. We used feedback from the LSI Logic toolset to restrict the class of circuits we admitted by changing the circuit recognizer to disallow any problem encountered. Later, we produced a much nicer circuit recognizer that produced error messages; the original recognizer only returned true or false. Another facet of the design process was ensuring that circuits could be tested. We implemented a parallel stuck-at fault simulator that was used during our design process and in the development of test vectors. Often during the design of the FM9001 logic, we would first check the testability of the circuit before we went to the effort to verify its correctness. We also used the fault simulator to identify redundant logic.

The FM9001 netlist, along with the necessary test vectors and signal pad assignments, was passed off to LSI Logic, Inc., at the end of July of 1991. We received FM9001 microprocessor chips some six weeks later. After receiving the chip, we built a very simple test jig that allowed any single instruction to be executed repeatedly so we could observe the pin-level timing of the chip. In 1992, we built a single-board computer using the FM9001 microprocessor; this board had two serial ports and a monitor program. This board was constructed in such a way that all of the FM9001 microprocessor signals could be attached to a logic analyzer. We executed programs by down-loading them from a workstation, and then observed the behavior of the FM9001 on our logic analyzer. We ran numerous programs to check to see if the manufactured part worked as we expected [1].

Our single-board FM9001 computer limited the testing we could perform; for instance, this single-board computer did not have a complete 2^{32} word memory. By early 1994, we had interfaced the FM9001 to a Tektronix chip tester; we then proceeded to test the FM9001 in all operational modes. We are able to generate random, pin-level inputs and check that the operation of the FM9001 chip perfectly matched what was predicted by the DUAL-EVAL simulator.[1] This testing has been completely automated. At this point, we have found no errors.

The FM9001 behavioral and design specifications, the hardware description language and its semantics, the large majority of the FM9001 correctness proof, and the preparation of the FM9001 for fabrication were performed entirely by Bishop Brock and Warren Hunt. Matt Kaufmann proved the ALU interpretation lemmas and assisted with proving the reset properties. Ann Siebert did much of the work on our recognizer for well-formed circuit descriptions that produces simple error messages, and Ken Albin built the FM9001 single-board computer.

III. An Introduction to DUAL-EVAL

DUAL-EVAL is the name of a function which we formally define below. DUAL-EVAL can be viewed both as a simulator and as a hardware description language. In fact, DUAL-EVAL was used to describe every gate-level primitive, wire, and input/output pad, required to physically fabricate the FM9001 microprocessor. DUAL-EVAL is an occurrence-oriented, hierarchical hardware description language developed to permit the hierarchical verification of Mealy machines. A predicate recognizes well-formed DUAL-EVAL circuit descriptions; this predicate requires a specific syntax without name clashes and it does not permit combinational loops. The operation of each circuit primitive is described by two functions: the outputs as a function of the inputs and internal state (the value specification) and the next internal state as a function of the

[1] There are two dedicated test pins for the level-sensitive register file that cannot be randomly set; we knew this prior to our testing. Also, the clock and power inputs cannot be randomly changed — the power supply must be stable and the clock input must be a suitable square wave.

inputs and internal state (the next-state specification)—"combinational circuits" only have one function, to compute the output. Circuits are generally verified in a hierarchical manner; that is, subcircuits of a larger circuit are first verified and then their specifications are composed to permit the verification of the larger circuit. In some cases where large random logic nests are verified, brute force comparison techniques are used, and in other cases where large repetitive circuits are required we verify functions that synthesize parameterized families of DUAL-EVAL netlists.

DUAL-EVAL is a formal simulator which is not tied to any particular hardware 'logic', e.g, four-valued or nine-valued. Nor is DUAL-EVAL tied to any particular set of primitive cells or gates. In general, the DUAL-EVAL user is free to define a hardware logic and appropriate primitives. For the F9001 project, we defined a four-valued logic and then a collection of primitive gates that operate on these four values. The primitives match a subset of LSI Logic's LCA10000 cell library, which we used in the FM9001 fabrication.

To verify a combinational circuit one only need verify that the value specification for a circuit is correct. However, for a Mealy machine both the value specification and the next-state specification must be proven correct. Circuit descriptions specified with DUAL-EVAL can be verified hierarchically and their specifications need not be confined to Boolean functions. For instance, it is possible to specify the contents of a bit-vector as a single arbitrarily-long list of Boolean variables instead of some specific number of Boolean variables. Thus, many of our Mealy machines specifications are extremely compact even though they may involve a large number of variables.

DUAL-EVAL computes two things when evaluating a module reference: (1) the outputs as a function of the inputs and the current state and (2) the next state, after one clock tick, as a function of the inputs and the current state. Even though our state-holding primitives, such as FD1, require a clock input, their definitions utterly ignores the clock input. The definition of our primitive state-holding devices specify both the output of the primitive, given its inputs and internal state, and what the next internal state should be when the clock "ticks."

Using a formally defined predicate that identifies well-formed netlists, we check that the same clock input is provided to the clock input of all state-holding primitives, thus requiring that all state-holding components update their internal states at the same time. For example, we check fanout and loading restrictions, delay times, well-formedness of names, etc. Each wire has a ptype, e.g., clock-type or Boolean. These types help insure that the clock input is only wired to the clock inputs of primitive state-holding devices. We do not further discuss these checks in this paper.

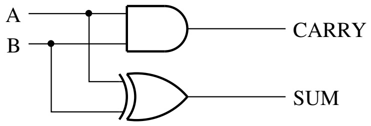


Fig. 1. Half-adder Schematic

IV. An Introduction to DUAL-EVAL Netlists

Prior to presenting the formal definition of DUAL-EVAL, we give a few examples of the primary data structures upon which DUAL-EVAL operates, namely (a) netlists and (b) the database of primitive modules. We also give a few examples of the execution of DUAL-EVAL on some sample netlists in the context of a database of primitives that is a small fragment of the database used in the FM9001 verification.

A well-formed circuit netlist is composed of well-formed circuit modules. Each module contains five elements: a module name, a list of input names, a list of output names, list of occurrences, and a list of occurrence names identifying those occurrences that contain state-holding devices.[2] A well-formed occurrence contains four elements: an occurrence name, a list of outputs, a reference to a module or primitive, and a list of inputs.

The following definition introduces a constant that is a DUAL-EVAL representation of a half-adder circuit. The name of the module is HALF-ADDER, its inputs are A and B, its outputs are SUM and CARRY, it has two occurrences, named G0 and G1, and finally, it contains no state-holding occurrences, hence the final NIL. The first occurrence passes the inputs A and B to the B-XOR primitive module, and the result of that primitive module becomes the value of the output SUM. The schematic for this module can be seen in Figure 1.

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

The following full-adder specification refers twice to the half-adder specification. Its schematic can be seen in Figure 2.

Definition.
```
(FULL-ADDER*)
```

[2]For a module containing only one state-holding occurrence, the single occurrence name may be used instead of a list containing that name. We later discuss this further when introducing the functions PAIRSTATES and COLLECT-STATES.

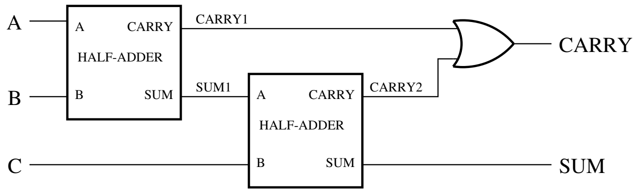


Fig. 2. Full-adder Schematic

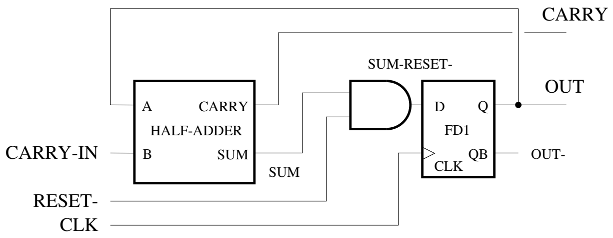


Fig. 3. One-Bit Counter Schematic

```
  =
'(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 now present a very simple one-bit synchronous counter with reset, which is shown in Figure 3. On each clock tick, this counter toggles its output `OUT` as long as `RESET-` is `T` and `CARRY-IN` is `T`. If `CARRY-IN` is `F`, then the output remains stable. If `RESET-` is `F`, then the counter is cleared, i.e., the output becomes `F` on the next clock tick.

Definition.

```
(ONE-BIT-COUNTER*)
  =
'(ONE-BIT-COUNTER
  (CLK CARRY-IN RESET-)
  (OUT CARRY)
  (( G0 (OUT OUT-)   FD1        (SUM-RESET- CLK))
   ( G1 (SUM CARRY)  HALF-ADDER (CARRY-IN OUT))
   ( G2 (SUM-RESET-) B-AND      (SUM RESET-)))
  G0)
```

We now combine four one-bit counters into a four-bit counter. The schematic drawing is shown in Figure 4. Here the state is represented as a list of occurrence names, while above in the one-bit counter, we only give the singleton occurrence name as there is only one state-holding element in that circuit.

Definition.

```
(FOUR-BIT-COUNTER*)
  =
```

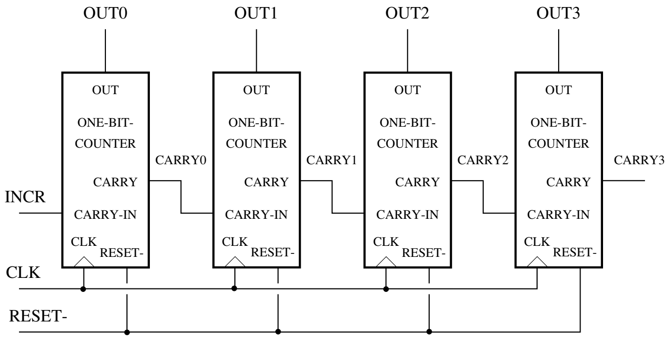


Fig. 4. Four-Bit Counter Schematic

```
'(FOUR-BIT-COUNTER
  (CLK INCR RESET-)
  (OUT0 OUT1 OUT2 OUT3)
  (( H0 (OUT0 CARRY0)
        ONE-BIT-COUNTER (CLK INCR   RESET-))
   ( H1 (OUT1 CARRY1)
        ONE-BIT-COUNTER (CLK CARRY0 RESET-))
   ( H2 (OUT2 CARRY2)
        ONE-BIT-COUNTER (CLK CARRY1 RESET-))
   ( H3 (OUT3 CARRY3)
        ONE-BIT-COUNTER (CLK CARRY2 RESET-)))
  (H0 H1 H2 H3))
```

We now give some example uses of `DUAL-EVAL`. Let us define:

Definition.

```
(EX-NETLIST)
  =
(LIST (HALF-ADDER*)
      (FULL-ADDER*)
      (ONE-BIT-COUNTER*)
      (FOUR-BIT-COUNTER*))
```

`DUAL-EVAL` requires five arguments: a flag indicating whether to compute the output or next state, the name of the module to be evaluated, the inputs to the module, the internal state of the module, and a netlist containing module definitions.

Our first example shows the computation of the output of the exclusive-or of `F` and `T`. The fourth argument is `NIL`, indicating no internal state; this is typical of combinational modules. The flag is `0`, indicating that we want the output.

```
(DUAL-EVAL 0 'B-XOR (LIST T F) NIL (EX-NETLIST))
  =
(LIST T)
```

We next evaluate the logical-and of `F` and `(X)`, our "unknown" value, which we formally introduce later. Note that given at least one `F`, the primitive `B-AND` always returns `F`, but with inputs `T` and `(X)`, the output is unknown.

```
(DUAL-EVAL 0 'B-AND (LIST F (X)) NIL (EX-NETLIST))
  =
(LIST F)


(DUAL-EVAL 0 'B-AND (LIST T (X)) NIL (EX-NETLIST))
  =
(LIST (X))
```

The following are example evaluations of the modules HALF-ADDER and FULL-ADDER. The result list is in the same order as the outputs defined in each module.

```
(DUAL-EVAL 0 'HALF-ADDER
           (LIST T F) NIL (EX-NETLIST))
  =
(LIST T F)


(DUAL-EVAL 0 'FULL-ADDER
           (LIST T F T) NIL (EX-NETLIST))
  =
(LIST F T)
```

We now consider evaluation of a synchronous circuit, ONE-BIT-COUNTER. To compute its output, we must supply, as always, its inputs and internal state, as shown below.

```
(DUAL-EVAL 0 'ONE-BIT-COUNTER
           (LIST F T T) T (EX-NETLIST))
  =
(LIST T T)
```

To compute the next state of our ONE-BIT-COUNTER module, we call DUAL-EVAL with 2 as the first argument.

```
(DUAL-EVAL 2 'ONE-BIT-COUNTER
           (LIST F T T) T (EX-NETLIST))
  =
F
```

We use the function SIMULATE to repeatedly apply DUAL-EVAL to the evaluation of a module. Note below that each element of the input list contains two values, the first one being a clock (whose value is utterly ignored). The output consists of a list of the value and next-state results.

```
(SIMULATE 'FOUR-BIT-COUNTER
          (LIST (LIST F F T)
                (LIST F F F)
                (LIST F F T)
                (LIST F F T)
                (LIST F T T)
                (LIST F T T)
                (LIST F T T)
                (LIST F T T))
          (LIST (X) (X) (X) (X))
          (EX-NETLIST))
  =
(LIST (LIST (LIST (X) (X) (X) (X))
            (LIST (X) (X) (X) (X)))
      (LIST (LIST (X) (X) (X) (X)) (LIST F F F F))
      (LIST (LIST F F F F)         (LIST F F F F))
      (LIST (LIST F F F F)         (LIST F F F F))
      (LIST (LIST F F F F)         (LIST T F F F))
      (LIST (LIST T F F F)         (LIST F T F F))
      (LIST (LIST F T F F)         (LIST T T F F))
      (LIST (LIST T T F F)         (LIST F F T F)))
```

V. Formal Presentation of DUAL-EVAL Semantics

In this section we formally define DUAL-EVAL in the Nqthm logic, cf. [6].

The first step in the definition of DUAL-EVAL is the definition of a database for the hardware logic and cell primitives that one wishes to use. DUAL-EVAL might have been defined to take this database as an argument, but in fact it was convenient to define it without such a parameter by "wiring-in" a specific database. The reader who is uninterested in this particular four-valued logic and our specific operations upon it may skip the next two subsections.

A. The Four-Valued Logic

Our first step is to define a four-valued logic, consisting of T (true), F (false), (Z) (floating), and (X) (unknown). We use two Nqthm "add-shell" events to introduce (X) and (Z) as two new distinct constants and the predicates XP and ZP, which recognize (X) and (Z), respectively.

The following fifteen functions are trivial operations on our four-valued logic. These functions recognize subsets of our logic, coerce values, and compute traditional operations such as "exclusive-or" and "or" over this logic. F-BUF coerces (Z) to (X) but is an identity function on T, F, and (X).

Definitions.

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


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


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


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

```
(F-BUF A)
  =
(THREEFIX A)


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


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


(F-NAND A B)
  =
(F-NOT (F-AND A B))


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


(F-NOR A B)
  =
(F-NOT (F-OR A B))


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


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


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


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


(F-PULLUP A)
  =
(IF (EQUAL A (Z))
    T
  (THREEFIX A))
```

B. *The* PRIMP-DATABASE

The evaluation of a DUAL-EVAL netlist eventually results in the interpretation of primitives, which are specified in by the definition of PRIMP-DATABASE. We here present a stripped-down version of the actual database we used in the FM9001 verification. We have reduced, for the purposes of presentation in this paper, both the number of primitives and the amount of information stored for each primitive. For example, the full entry for B-AND is actually:

```
(B-AND (DELAYS ((144 422) (54 707)))
       (DRIVES 10)
       (INPUT-TYPES BOOLP BOOLP)
       (INPUTS A B)
       (LOADINGS 1 1)
       (LSI-NAME . AN2)
       (OUT-DEPENDS (A B))
       (OUTPUT-TYPES BOOLP)
       (OUTPUTS Z)
       (RESULTS CONS (F-AND A B) 'NIL)
       (GATES . 2)
       (PRIMITIVES . 1)
       (TRANSISTORS . 6))
```

However, in the definition of PRIMP-DATABASE-SMALL below, we only list the properties directly required by DUAL-EVAL. The actual PRIMP-DATABASE used in the FM9001 verification is more than 1200 lines long. The other properties are used for such matters as calculating gate delays, combinational loops, and drive strength. An entry in the database associates with a name, e.g., FD1, certain values for the properties INPUTS, RESULTS, and in the case of state holding primitives, NEW-STATES and STATES. The INPUTS property merely specifies the names of the formal input parameters to the primitive. Likewise, for state-holding primitives, the property STATES names the one internal state component. The RESULTS property expresses the value produced by this primitive. This value is presented as an expression to be interpreted by the Nqthm evaluator function EVAL$, which is closely akin to Lisp's EVAL in generality. The result of such an evaluation is always a single list whose length is that of the number of outputs of the primitive. Similarly, for state-holding primitives, the NEW-STATES property is an expression for calculating the new value of the internal state. Thus, we can see from the following database that FD1 is a primitive with two inputs named D and CP, an internal state named STATE, and two output lines whose values are (F-BUF STATE) and (F-NOT STATE) respectively. The "next value" of STATE after each "clock-tick" is (F-BUF D). The primitive FD1SLP is like FD1 but provides for a scan chain.

Definition.

```
(PRIMP-DATABASE-SMALL)
  =
'((B-AND (INPUTS A B)
         (RESULTS . (CONS (F-AND A B) 'NIL)))
  (B-OR (INPUTS A B)
        (RESULTS . (CONS (F-OR A B) 'NIL)))
  (B-NAND (INPUTS A B)
          (RESULTS . (CONS (F-NAND A B) 'NIL)))
  (B-NOR (INPUTS A B)
         (RESULTS . (CONS (F-NOR A B) 'NIL)))
  (B-XOR (INPUTS A B)
         (RESULTS . (CONS (F-XOR A B) 'NIL)))
  (FD1 (INPUTS D CP)
       (NEW-STATES . (F-BUF D))
       (RESULTS . (CONS
                   (F-BUF STATE)
                   (CONS (F-NOT STATE) 'NIL)))
       (STATES . STATE))
  (FD1SLP (INPUTS D CP LD TI TE)
          (NEW-STATES . (F-IF TE TI
                              (F-IF LD D STATE)))
          (RESULTS . (CONS
                      (F-BUF STATE)
                      (CONS (F-NOT STATE) 'NIL)))
          (STATES . STATE))
  (ID (INPUTS A)
      (RESULTS . (CONS A 'NIL)))
  (T-BUF (INPUTS E A)
         (RESULTS . (CONS (FT-BUF E A) 'NIL)))
  (T-WIRE (INPUTS A B)
          (RESULTS . (CONS (FT-WIRE A B) 'NIL)))
  (PULLUP (INPUTS A)
          (RESULTS . (CONS (F-PULLUP A) 'NIL))))
```

The primitive ID is a renaming primitive. The primitives T-BUF, T-WIRE, and PULLUP are used for modeling tri-state logic circuits. We used primitives similar to these to model the bi-directional data bus for the FM9001.

C. The Formal Definition of DUAL-EVAL

The function VALUE takes as arguments a NAME and an association list ALIST and returns the value associated with NAME in ALIST, if any, and returns 0 otherwise. COLLECT-VALUE takes a list of names ARGS and an association list ALIST and return the list of the values of the member of ARGS with respect to ALIST.

Definitions.

```
(VALUE NAME ALIST)
  =
(IF (NLISTP ALIST)
    0
  (IF (LISTP (CAR ALIST))
      (IF (EQUAL (CAAR ALIST) NAME)
          (CDAR ALIST)
        (VALUE NAME (CDR ALIST)))
    (VALUE NAME (CDR ALIST))))


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


(PAIRSTATES STATENAMES STATE)
  =
(IF (OR (LISTP STATENAMES)
        (EQUAL STATENAMES NIL))
    (PAIRLIST STATENAMES STATE)
  (LIST (CONS STATENAMES STATE)))


(COLLECT-STATES STATENAMES STATE-BINDINGS)
  =
(IF (OR (LISTP STATENAMES)
        (EQUAL STATENAMES NIL))
    (COLLECT-VALUE STATENAMES STATE-BINDINGS)
  (VALUE STATENAMES STATE-BINDINGS))
```

The function PAIRSTATES associates a list of state names with an equal number of state values. Singletons are treated specially. Thus

```
(PAIRSTATES 'S 3) = '((S . 3))  and

(PAIRSTATES '(S U) '(1 2)) = '((S . 1) (U . 2))
```

The function COLLECT-STATES takes a list STATENAMES and an association list STATE-BINDINGS and produces the list of the values of the members of STATENAMES with respect to STATE-BINDINGS. Singletons are treated specially. Thus

```
(COLLECT-STATES '(S1 S2)
                '((S3 . 33) (S2 . 22) (S1 . 11)))
  =
'(11 22)  and

(COLLECT-STATES 'S3
                '((S3 . 33) (S2 . 22) (S1 . 11)))
  =
33
```

The function DELETE-MODULE deletes an entry associated with NAME from the association list NETLIST. The function LOOKUP-MODULE is like VALUE except that F rather than 0 is the default. PRIMP is used to look up a definition of primitive FN in (PRIMP-DATABASE-SMALL). PRIMP-LOOKUP returns the property NAME associated with the primitive FN in (PRIMP-DATABASE-SMALL).

Definitions.

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


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


(PRIMP FN)
  =
(LOOKUP-MODULE FN (PRIMP-DATABASE-SMALL))


(PRIMP-LOOKUP FN NAME)
  =
(LOOKUP-MODULE NAME (CDR (PRIMP FN)))


(PRIMP2 FN NAME)
  =
(CDR (PRIMP-LOOKUP FN NAME))
```

The following five functions merely select pieces of a module. The next four functions select pieces of an occurrence.

Definitions.

```
(MODULE-NAME        MODULE) = (CAR MODULE)
(MODULE-INPUTS      MODULE) = (CADR MODULE)
(MODULE-OUTPUTS     MODULE) = (CADDR MODULE)
(MODULE-OCCURRENCES MODULE) = (CADDDR MODULE)
(MODULE-STATENAMES  MODULE) = (CADDDDR MODULE)

(OCC-NAME     OCCURRENCE) = (CAR OCCURRENCE)
(OCC-OUTPUTS  OCCURRENCE) = (CADR OCCURRENCE)
(OCC-FUNCTION OCCURRENCE) = (CADDR OCCURRENCE)
(OCC-INPUTS   OCCURRENCE) = (CADDDR OCCURRENCE)


(DUAL-APPLY-VALUE FN ARGS STATE)
  =
(LET ((INPUT-NAMES (PRIMP2 FN 'INPUTS))
      (STATE-NAMES (PRIMP2 FN 'STATES))
      (MODULE-OUTS (PRIMP2 FN 'RESULTS)))
  (LET ((ALIST
         (APPEND (PAIRLIST INPUT-NAMES ARGS)
                 (PAIRSTATES STATE-NAMES STATE))))
    (EVAL$ T MODULE-OUTS ALIST)))


(DUAL-APPLY-STATE FN ARGS STATE)
  =
(LET ((INPUT-NAMES (PRIMP2 FN 'INPUTS))
      (STATE-NAMES (PRIMP2 FN 'STATES))
      (NEW-STATES  (PRIMP2 FN 'NEW-STATES)))
  (LET ((ALIST
         (APPEND (PAIRLIST INPUT-NAMES ARGS)
                 (PAIRSTATES STATE-NAMES STATE))))
    (EVAL$ T NEW-STATES ALIST)))
```

The function DUAL-APPLY-VALUE given primitive name FN, inputs ARGS, and internal states STATE computes the output values for the primitive. The function DUAL-APPLY-STATE given primitive name FN, inputs ARGS, and internal state STATE, computes the next internal state for the primitive.

DUAL-EVAL may be regarded as four mutually recursive functions defined as a single function, with the first parameter FLAG, ranging from 0 to 3, indicating which of the four functions is really intended. (The Nqthm logic does not permit the definition of mutually recursive functions.) For each of the four values of FLAG we now roughly characterize what DUAL-EVAL computes.

- If FLAG is 0, then compute the outputs of the module X0 with inputs X1 and internal state X2, given either that the module is a primitive or it is defined in NETLIST.
- If FLAG is 1, then compute the value of each occurrence in X0 with respect to the input and state values in the association lists X1 and X2.
- If FLAG is 2, then compute the next state for module X0 with inputs X1 and internal state X2.
- If FLAG is 3, then compute the next state of each occurrence in X0 with respect to the input and state values in the association lists X1 and X2.

If DUAL-EVAL is called with FLAG either 0 or 1, then in all subsidiary recursive calls of DUAL-EVAL, FLAG remains either 0 or 1. The calculation of the output values is independent of the calculation of the next state. However, when DUAL-EVAL is called with FLAG being either 2 or 3, then recursive calls with FLAG taking on any of the four values is possible because the calculation of the next state is, in general, dependent upon the output values of subsidiary modules. In normal use, DUAL-EVAL is called with the FLAG argument 0 or 2, and with argument X0 set to the name of the module to be evaluated. When FLAG is 1 or 3, argument X0 is set to a list of occurrences. Argument X1 always contains inputs; when FLAG is 0 or 2, it contains an input list, and when FLAG is 1 or 3, it contains an association list mapping names to values. Argument X2 is used in a similar fashion for the state-holding arguments.

Definition.

```
(DUAL-EVAL FLAG X0 X1 X2 NETLIST)
  =
(CASE FLAG

 ;; New value of a form.

(0 (LET ((FN X0)
         (ARGS X1)
         (STATE X2))
     (IF (PRIMP FN)
         (DUAL-APPLY-VALUE FN ARGS STATE)

         (LET ((MODULE (LOOKUP-MODULE FN NETLIST)))
           (IF (LISTP MODULE)
               (LET
                 ((INPUTS  (MODULE-INPUTS MODULE))
                  (OUTPUTS (MODULE-OUTPUTS MODULE))
                  (OCCURRENCES (MODULE-OCCURRENCES MODULE))
                  (STATENAMES (MODULE-STATENAMES MODULE)))
                 (COLLECT-VALUE
                  OUTPUTS
                  (DUAL-EVAL 1
                             OCCURRENCES
                             (PAIRLIST INPUTS ARGS)
                             (PAIRSTATES STATENAMES STATE)
                             (DELETE-MODULE FN NETLIST))))
               F)))))

 ;; Create the new bindings of a body.

(1 (LET ((BODY X0)
         (BINDINGS X1)
         (STATE-BINDINGS X2))

     (IF (LISTP BODY)
         (LET ((OCCURRENCE (CAR BODY)))
           (LET ((OCC-NAME (OCC-NAME OCCURRENCE))
                 (OUTPUTS (OCC-OUTPUTS OCCURRENCE))
                 (FN (OCC-FUNCTION OCCURRENCE))
                 (INPUTS (OCC-INPUTS OCCURRENCE)))
             (DUAL-EVAL
              1
              (CDR BODY)
              (APPEND
               (PAIRLIST OUTPUTS
                         (DUAL-EVAL
                          0
                          FN
                          (COLLECT-VALUE INPUTS BINDINGS)
                          (VALUE OCC-NAME STATE-BINDINGS)
                          NETLIST))
               BINDINGS)
              STATE-BINDINGS
              NETLIST)))
         BINDINGS)))

 ;; New state of a form.

(2 (LET ((FN X0)
         (ARGS X1)
         (STATE X2))
     (IF (PRIMP FN)
         (DUAL-APPLY-STATE FN ARGS STATE)

         (LET ((MODULE (LOOKUP-MODULE FN NETLIST)))
           (IF (LISTP MODULE)
               (LET
                 ((INPUTS  (MODULE-INPUTS MODULE))
                  (OUTPUTS (MODULE-OUTPUTS MODULE))
                  (OCCURRENCES (MODULE-OCCURRENCES MODULE))
                  (STATENAMES (MODULE-STATENAMES MODULE)))
                 (COLLECT-STATES
                  STATENAMES
                  (DUAL-EVAL
                   3
                   OCCURRENCES
                   (DUAL-EVAL 1
                              OCCURRENCES
                              (PAIRLIST INPUTS ARGS)
                              (PAIRSTATES STATENAMES STATE)
                              (DELETE-MODULE FN NETLIST))
                   (PAIRSTATES STATENAMES STATE)
                   (DELETE-MODULE FN NETLIST))))
               F)))))

 ;; New state of a body.

(3 (LET ((BODY X0)
         (BINDINGS X1)
         (STATE-BINDINGS X2))

     (IF (LISTP BODY)
         (LET ((OCCURRENCE (CAR BODY)))
           (LET ((OCC-NAME (OCC-NAME OCCURRENCE))
                 (OUTPUTS (OCC-OUTPUTS OCCURRENCE))
                 (FN (OCC-FUNCTION OCCURRENCE))
                 (INPUTS (OCC-INPUTS OCCURRENCE)))
             (CONS
              (CONS OCC-NAME
                    (DUAL-EVAL
                     2
                     FN
                     (COLLECT-VALUE INPUTS BINDINGS)
                     (VALUE OCC-NAME STATE-BINDINGS)
                     NETLIST))
              (DUAL-EVAL 3
                         (CDR BODY)
                         BINDINGS
                         STATE-BINDINGS
                         NETLIST))))
         NIL)))

(OTHERWISE F))
```

The termination proof required by the Nqthm definition principle for DUAL-EVAL hinges upon the fact that we delete each module from the netlist as it is evaluated. In other words, we do not support recursive modules.

DUAL-EVAL was designed to make hierarchical verification of Mealy machines possible. Each time a module is specified, only two lemmas need to be proven: a lemma specifying the module's output and a lemma specifying the module's next state. These lemmas are used to prove the correctness of yet larger modules containing these submodules, without the need to delve into any further details about the submodules.

The function SIMULATE provides a mechanism for the repeated application of DUAL-EVAL, thus providing a multi-step Mealy machine evaluator. SIMULATE returns the trace of the operation of the module as a list of pairs of the value and next state for each step. The argument INPUTS provides external inputs for each step for module FN and the argument STATE provides the initial state. The argument NETLIST provides the definition for each module. The function SIMULATE-DUAL-EVAL-2 returns the final internal state after N steps.

Definitions.

```
(SIMULATE FN INPUTS STATE NETLIST)
  =
(IF (NLISTP INPUTS)
    NIL
  (LET ((VALUE (DUAL-EVAL 0 FN (CAR INPUTS)
                          STATE NETLIST))
        (NEW-STATE (DUAL-EVAL 2 FN (CAR INPUTS)
                              STATE NETLIST)))
    (CONS (LIST VALUE NEW-STATE)
          (SIMULATE FN (CDR INPUTS)
                    NEW-STATE NETLIST))))


(SIMULATE-DUAL-EVAL-2 MODULE INPUTS STATE
                      NETLIST N)
  =
(IF (ZEROP N)
    STATE
  (SIMULATE-DUAL-EVAL-2
   MODULE
   (CDR INPUTS)
   (DUAL-EVAL 2 MODULE (CAR INPUTS) STATE NETLIST)
   NETLIST
   (SUB1 N)))
```

VI. Some Example DUAL-EVAL Theorems

As mentioned previously, we always prove a value lemma for a combinational circuit and both value and state lemmas for a sequential circuit. Here, for example, is the value lemma for the HALF-ADDER circuit.

Theorem.

```
(IMPLIES
 (EQUAL (LOOKUP-MODULE 'HALF-ADDER NETLIST)
        (HALF-ADDER*))
 (EQUAL (DUAL-EVAL 0 'HALF-ADDER (LIST A B)
                   STATE NETLIST)
        (LIST (F-XOR A B)
              (F-AND A B))))
```

Here is the value lemma for the FULL-ADDER circuit. Its proof makes use of the previous HALF-ADDER lemma.

Theorem.

```
(IMPLIES
 (AND
  (EQUAL
   (LOOKUP-MODULE 'HALF-ADDER NETLIST)
   (HALF-ADDER*))
  (EQUAL
   (LOOKUP-MODULE 'FULL-ADDER NETLIST)
   (FULL-ADDER*)))
 (EQUAL (DUAL-EVAL 0 'FULL-ADDER (LIST A B C)
                   STATE NETLIST)
        (LIST (F-XOR (F-XOR A B) C)
              (F-OR (F-AND A B)
                    (F-AND (F-XOR A B) C)))))
```

For our simple, sequential, one-bit counter circuit, we prove two lemmas, the value and the next-state lemma. These two lemmas are shown below.

Theorem.

```
(IMPLIES
 (AND
  (EQUAL (LOOKUP-MODULE 'HALF-ADDER NETLIST)
         (HALF-ADDER*))
  (EQUAL (LOOKUP-MODULE 'ONE-BIT-COUNTER NETLIST)
         (ONE-BIT-COUNTER*)))
 (EQUAL (DUAL-EVAL 0 'ONE-BIT-COUNTER
                   (LIST CLK CARRY-IN RESET-)
                   STATE NETLIST)
        (LIST (F-BUF STATE)
              (F-AND CARRY-IN (F-BUF STATE)))))
```

Theorem.

```
(IMPLIES
 (AND
  (EQUAL (LOOKUP-MODULE 'HALF-ADDER NETLIST)
         (HALF-ADDER*))
  (EQUAL (LOOKUP-MODULE 'ONE-BIT-COUNTER NETLIST)
         (ONE-BIT-COUNTER*)))
 (EQUAL (DUAL-EVAL 2 'ONE-BIT-COUNTER
                   (LIST CLK CARRY-IN RESET-)
                   STATE NETLIST)
        (F-AND RESET-
               (F-XOR CARRY-IN (F-BUF STATE)))))
```

We now show the lemmas that we prove to demonstrate the correctness of our four-bit counter. First, however, we define two functions:

Definitions.

```
(V-INCR C A)
  =
(IF (NLISTP A)
    NIL
  (CONS (F-XOR C (CAR A))
        (V-INCR (F-AND C (CAR A))
                (CDR A))))


(V-CLEAR C A)
  =
(IF (NLISTP A)
    NIL
  (CONS (F-AND C (CAR A))
        (V-CLEAR C (CDR A))))
```

We now present the value and next-state lemmas for our our four-bit counter circuit. Note that the proof of these two lemmas do not involve reasoning about the internals of the one-bit counter circuits, but use the value and next-state lemmas given above for ONE-BIT-COUNTER.

Theorem.
```
(IMPLIES
 (AND
  (EQUAL (LOOKUP-MODULE 'HALF-ADDER NETLIST)
         (HALF-ADDER*))
  (EQUAL (LOOKUP-MODULE 'ONE-BIT-COUNTER NETLIST)
         (ONE-BIT-COUNTER*))
  (EQUAL (LOOKUP-MODULE 'FOUR-BIT-COUNTER NETLIST)
         (FOUR-BIT-COUNTER*)))
 (EQUAL (DUAL-EVAL 0 'FOUR-BIT-COUNTER
                   (LIST CLK INCR RESET-)
                   (LIST S0 S1 S2 S3) NETLIST)
        (LIST (F-BUF S0)
              (F-BUF S1)
              (F-BUF S2)
              (F-BUF S3))))
```

Theorem.
```
(IMPLIES
 (AND
  (EQUAL (LOOKUP-MODULE 'HALF-ADDER NETLIST)
         (HALF-ADDER*))
  (EQUAL (LOOKUP-MODULE 'ONE-BIT-COUNTER NETLIST)
         (ONE-BIT-COUNTER*))
  (EQUAL (LOOKUP-MODULE 'FOUR-BIT-COUNTER NETLIST)
         (FOUR-BIT-COUNTER*)))
 (EQUAL (DUAL-EVAL 2 'FOUR-BIT-COUNTER
                   (LIST CLK INCR RESET-)
                   (LIST S0 S1 S2 S3) NETLIST)
        (V-CLEAR RESET-
                 (V-INCR INCR
                         (LIST S0 S1 S2 S3)))))
```

The FM9001 verification required proving such value and next-state lemmas for every internal module in the FM9001 microprocessor. All of the lemmas presented above are extremely trivial. We present them in this paper, however, to emphasize that we are concerned with the mechanical checking of mathematical statements about our DUAL-EVAL-defined circuits. The sequence of definitions and lemmas leading to the principal result about the FM9001, described in Section VII below, consists of more than 1500 definitions and 2000 lemmas occupying more than 2 megabytes of mechanically checked text. For all the details about these definitions and lemmas, see the report "The FM9001 Microprocessor Proof," [10].

The main steps involved in proving the principal theorem about the FM9001 include the formulation and proof of the following collection of concepts and theorems.

- Logical bureaucracy. Developed techniques to guide the mechanical theorem-prover to efficiently finding proofs about natural number and integer operations and EVAL$ computations.
- Data abstraction. Constructed mappings between bit-vectors, integers, and natural numbers.
- Functional abstraction. Specified adders, shifters, register files, and multiplexers, out of gate-level operations.
- Interface mechanisms. Built external communication protocols necessary for the tri-state memory interface.
- Complexity management. Constructed verified hardware synthesis programs that produce guaranteed correct n-bit wide combinational and sequential circuits for arbitrary n.
- Control logic abstraction. Created a means to generate control logic automatically from symbolic representations of internal control algorithms.
- Time abstraction. Defined a mapping between machine-code instructions and single clock microcode events.
- Composition of time and data abstraction. Formally connected the FM9001 to a suitable memory, with provision for arbitrary memory delays and then proved that the entire FM9001 implementation and memory system operates exactly as specified at the behavioral level.

VII. A DUAL-EVAL Theorem Stating the Correctness of the FM9001

We believe that a significant result of the FM9001 microprocessor study is that it is possible to formally model a microprocessor at several levels of abstraction and then prove that these different levels are formally related. The user-level specification of the FM9001 is specified operationally as an interpreter, a function named FM9001, that transforms a user-level state (i.e., the memory, registers, and flags) by the execution of n instructions [24]. This is shown as the lower arrow, named **Boolean Interpretation**, at the top of Figure 5. The **Natural Number Interpretation** and the **Integer Interpretation** define the operation of the FM9001 when the memory and register words are viewed as natural numbers and integers, respectively. These two interpretations have been proved to be equivalent to the **Boolean Interpretation**. Thus, irrespective of the actual implementation, the user-level specification has been shown to correctly implement natural number and integer arithmetic; these interpretations are the basis for the correctness of the Piton assembler [31] for the FM9001.

The FM9001 is specified at four levels of abstraction—user, two-valued, four-valued, and netlist; these are shown in Figure 5.

- A high-level behavioral (user-level) model that operates as an instruction interpreter for FM9001 programs, the function FM9001; this function is a formal

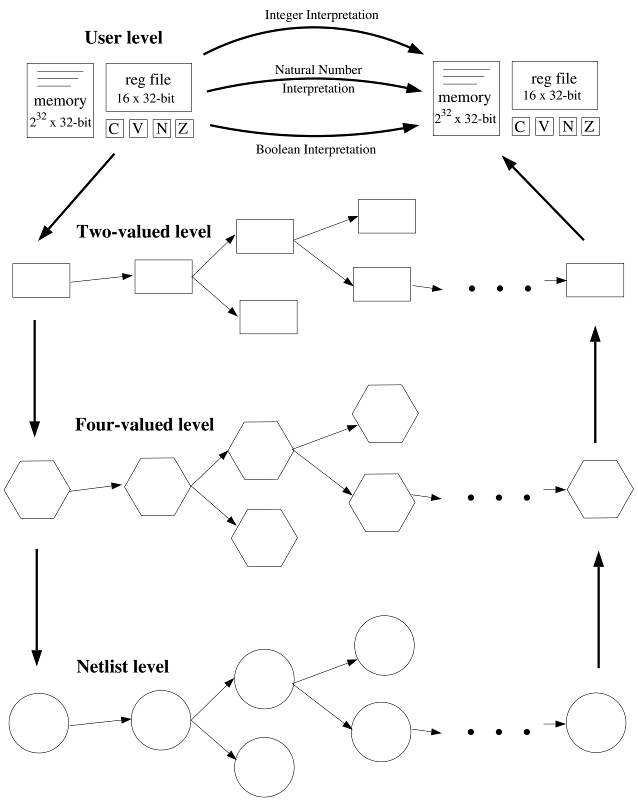


Fig. 5. Specification Levels

version of a machine-code programmer's view of the FM9001.

- A Boolean (two-valued) model that "mirrors" the intermediate, register-transfer model.
- An intermediate, (four-valued) register-transfer model that links the low- and high-level specifications for verification purposes.
- A complete gate-level (netlist) model presented in the `DUAL-EVAL` HDL.

The high-level and the register-transfer models are captured as executable Nqthm logic functions. The gate-level implementation, which is described by an Nqthm constant, is given semantics by our hardware description language simulator, `DUAL-EVAL`. Even though we often prove that parts of the register-transfer level are equivalent to the "Boolean level", we do not have a complete Boolean level; this is an artifact of our verification approach. The gate-level model includes all of the logic required to physically construct the FM9001, including the test logic. To have LSI Logic manufacture the FM9001, we translated only the `DUAL-EVAL` netlist into LSI Logic's Network Description Language (NDL) and provided the test vectors required by LSI Logic, Inc.

The proof of correctness of the FM9001 gate-level design consists of three major lemmas. First, we have shown that the FM9001 can be forced to a known state, i.e., reset, by a suitable sequence of inputs. This proof is carried out using the gate-level models; the concept of resetting the machine does not appear at the behavioral level of the instruction interpreter. Second, we prove that given a set of initial conditions, the gate-level model correctly implements the high-level specification, cf. theorem `FM9001=CHIP-SYSTEM-SUMMARY`. This proof involves both time and data abstractions to link the clock-cycle based semantics of the gate-level FM9001 model with the high-level instruction interpreter. Finally, we show that the state at the end of the reset sequence satisfies the initial conditions for the previous lemma. This sequence of results provides the formal basis for believing that behavior specified by the instruction interpreter is implemented by the gate-level design. Of course, any physical flaws could invalidate our assumptions. We did, during the design of the FM9001, make provisions for thoroughly testing the manufactured devices.

The FM9001 executes instructions by fetching operand A, operand B if necessary, computes a result, and stores the result in the location referenced by operand B, if necessary. We describe the instruction set more thoroughly in Section VIII. The top of Figure 6 shows the two FM9001 instruction formats, a two-address instruction or an immediate datum instruction.

The FM9001 programmer's state is shown as a part of the user level in Figure 5. This state includes a general-purpose 16-element register file, four arithmetic flags, and the external memory. Register 15 is usually used for 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 actually allows any of the 16 registers to be used as the program counter.

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. The netlist-level specification describes the actual Boolean gates and latches, test logic, and I/O buffers used to implement the FM9001.

Before giving an informal statement of our chief result about the FM9001, we describe that statement in terms of Figure 5. The user-level operation of the FM9001 is shown at the top by arrows transforming a memory, register file, and flag state to a new user-level state, which results from executing one user-level instruction. The correctness of the implementation is stated by mapping a user-level state down to a netlist-level state (see the lower left corner), executing the `DUAL-EVAL` simulator on that state with the FM9001 netlist-level implementation, and then mapping the resultant state back up to the same final user-level state. Further, the figure is drawn to indicate that more steps are required to implement an instruction at the lower levels than at the user level.

Here is an informal statement of the chief result proved about the FM9001. Let `H` be the hardware netlist that we have constructed for the FM9001 with the constant `CHIP-SYSTEM$NETLIST`. We have proved that for each well-formed FM9001 user-visible state, `S`, as checked by the predicate `FM9001-STATEP`, and for each positive integer, `N`, there exists a positive integer `C` such that the result of running the user-model of the FM9001 (the function `FM9001`) `N` steps can instead be obtained by simulating `H` and `S` under the `DUAL-EVAL` semantics for `C` steps. In precisely stating the theorem, we arrange to supply the `DUAL-EVAL` semantics with a transform (obtained with `MAP-DOWN`) of `S`, and afterwards we do a reverse transformation (with `MAP-UP`) to obtain the final user-visible state. Furthermore, in making the statement precise we stipulate that the external stimuli, `I`, to the chip do not enable a hold, a reset, or a test input for the `C` clock cycles, as checked by the predicate `NO-HOLDS-RESET-OR-TEST`. The fact that `N` and `C` are different reflects the fact that a single FM9001 instruction takes several clock cycles to emulate at the `DUAL-EVAL` netlist level. The precise statement of this correctness result is:

Theorem `FM9001=CHIP-SYSTEM-SUMMARY.`

```
 (LET ((H (CHIP-SYSTEM$NETLIST))
       (C (TOTAL-MICROCYCLES (MAP-DOWN S)
                             (MAP-UP-INPUTS I) N)))
   (IMPLIES
    (AND (FM9001-STATEP S)
         (NO-HOLDS-RESET-OR-TEST I C))
    (EQUAL
     (FM9001 S N)
     (MAP-UP
      (SIMULATE-DUAL-EVAL-2 'CHIP-SYSTEM I
                            (MAP-DOWN S) H C)))))
```

The above theorem is a verification of the complete FM9001 netlist, including all I/O pads, the bi-directional memory bus, and all internal logic.

The proof of correctness of the FM9001 microprocessor model was mechanically checked by the Nqthm theorem prover. Our FM9001 proof script contains 2957 entries that expand into 4851 theorem-prover events. The total time required for Nqthm to check the proof of the FM9001's implementation with respect to its specification is about 4 hours on a SPARCstation 10. The statement of the three correctness lemmas takes almost 10,000 lines to "prettyprint".

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

TABLE I
Size of FM9001 proof statement.

Given the size of the formula that we needed to manipulate, we considered it necessary to have mechanical assistance. Our choice of the Nqthm system was based on our belief that it could successfully discharge the rather large proof obligations we faced during our effort.

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, in principle, 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 [19] 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 definition of the FM9001 architecture, its design, the definition of the `DUAL-EVAL` netlist simulator, the verification of the three lemmas mentioned above, generation of the test vectors, and the conversion of the FM9001 `DUAL-EVAL` netlist to LSI Logic's NDL, required about three man-years of effort. Approximately an additional six months of effort was devoted to testing the manufactured FM9001. The time required to document this effort is not included in the estimates just given.

The full formal details of this theorem may be found in the technical report "The FM9001 Microprocessor Proof," [10].

VIII. The FM9001 User-level Instruction Set

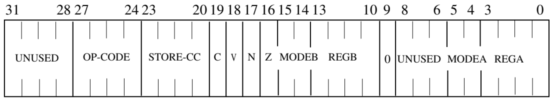
We now present a short, informal description of the instruction set of the processor, and the pinout of the actual chip. All of our formal specifications represent varying degrees of abstraction about the behavior of the physical device. This preliminary introduction of the FM9001 is a foundation on which to build the formal specifications and proofs.

The FM9001 is a general-purpose 32-bit microprocessor featuring a two-address programming architecture with five addressing modes, 16 general-purpose 32-bit registers, 15 arithmetic and logical operations, extensive arithmetic flag support, and a memory interface that supports multiple processors. Figure 6 shows the two FM9001 instruction formats, a two-address instruction and an immediate datum instruction. The two formats only differ in the origin of operand A. In the immediate datum case the nine-bit datum in the instruction word is sign-extended to 32-bits. Otherwise, operand A is selected in the same manner as operand B. Except for the immediate datum mode, operand A and operand B are selected by any of the following modes: register direct, register indirect, register indirect with pre-increment, and register indirect with post-increment. Thus, operand A has five addressing modes and operand B has four, and every addressing mode for both operand A and B works with every instruction. Except for the immediate addressing mode for operand A there is no provision for data sizes other than 32-bit words.

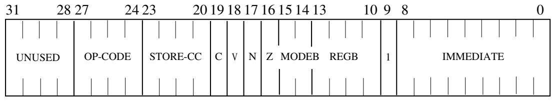
One of the general-purpose registers is used as the program counter, and the choice of the PC is programmable externally. The number of the register used as the PC is read from dedicated pins during reset and hold operations on the FM9001. The PC number can be neither read nor written by any instruction. This feature provides a very primitive interrupt facility: The processor can be halted by a hold request, the PC number changed, and then processing resumed on a different instruction stream. Since branching instructions and certain types of immediate data instructions depend on knowledge of which register is being used as the PC, programs written for the FM9001 will depend on assumptions about the hardware and execution environment of the FM9001. Generally, we select register 15 as the program counter.

The 15 different arithmetic and logical instructions are listed in Figure 6, as are the conditions that determine if the computed result is to be stored. Each instruction also contains four bits that determine individually whether the arithmetic flags carry, overflow, zero, and negative, are updated.[3] During instruction execution the PC is incremented by 1 before the calculation of the effective address of the A operand, and all side effects to the A operand register (pre-decrement or post-increment) are completed before the computation of the effective address of the B operand.

TWO-ADDRESS MODE



IMMEDIATE DATUM MODE



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	MNEMONIC	OPERATION	STORE-CC	MNEMONIC	CONDITION
0000	MOVE	b <- a	0000	CC	~C
0001	INC	b <- a + 1	0001	CS	C
0010	ADDC	b <- a + b + c	0010	VC	~V
0011	ADD	b <- b + a	0011	VS	V
0100	NEG	b <- 0 - a	0100	PL	~N
0101	DEC	b <- a - 1	0101	MI	N
0110	SUBB	b <- b - a - c	0110	NE	~Z
0111	SUB	b <- b - a	0111	EQ	Z
1000	ROR	b <- c,a >> 1	1000	HI	~C & ~Z
1001	ASR	b <- a >> 1	1001	LS	C | Z
1010	LSR	b <- a >> 1	1010	GE	(N & V) | (~N & ~V)
1011	XOR	b <- b XOR a	1011	LT	(N & ~V) | (~N & V)
1100	OR	b <- b OR a	1100	GT	(N & V & ~Z) | (~N & ~V & ~Z)
1101	AND	b <- b AND a	1101	LE	Z | (N & ~V) | (~N & V)
1110	NOT	b <- NOT a	1110	T	True
1111	M15	b <- a	1111	F	False

Fig. 6. FM9001 Instruction Word Format

Some simple examples will help clarify some of the features of the architecture and instruction set. Since the results of any instruction may be conditionally stored, and the register used as the PC is always valid as a destination, there are no differences between control and data operations. For example a "Jump" instruction can be encoded as an unconditional MOVE (i.e., STORE-CC = 1110) of the target address to the PC. A conditional jump is obtained simply by encoding a condition other than "True" in the STORE-CC field of the instruction. For example "Jump if Carry clear" is encoded as a MOVE instruction where the target address is the PC, with STORE-CC = 0000. Non-storing instructions (i.e., STORE-CC = 1111) can be used to side effect the flags without destroying the destination. For example, a "Compare" can be effected by a non-storing SUB that sets the condition codes (OP-CODE = 0111, CVNZ=1111, STORE-CC=1111).

The FM9001 specification was derived from the FM8502 specification [23]. There are several important differences.

- The FM8502 reserves register 15 for the PC. The PC is hardware selectable in the FM9001.
- The FM9001 has the ability to conditionally store every result. Only conditional moves were permitted

[3] This feature was added to the FM8502 processor to simplify the proofs of correctness of the PITON assembler [31]. We retained this feature in the FM9001.

in the FM8502.

- The FM9001 performs the post-increment operations as each operand is fetched. The FM8502 post-increment operations were done after both operands were fetched and the result was stored.

- The FM9001 has an immediate data mode for operand A; the FM8502 did not have this feature.

The ALU operations remain the same as do the fetch-execute cycle. We did this to ensure that the Piton assembler [32] for the FM8502 could be easily transported to the FM9001.

IX. The Physical FM9001 Integrated Circuit

Physically, the FM9001 is a general-purpose CMOS 32-bit microprocessor implemented as a gate array. The FM9001 features a two-address programming architecture, five addressing modes, multi-processor bus capability, and extensive arithmetic flag support. Assuming a fast enough external memory, instructions require from five to 17 cycles to execute; typical instructions require five to nine clock cycles to execute. At 25 MHz the FM9001 only requires 70 mA of power exclusive of I/O drivers; the static design allows the clock rate to be reduced for lower power consumption. All internal flip-flops are connected by a scan chain for testing purposes, except for the register-file latches which have dedicated test signals. We have been testing the FM9001 chips intermittently for three years, and we have never found a case where the behavior of FM9001 chips deviates from its specifications.

A pictorial diagram of the FM9001 signal groups appears as Figure 7. The implementation of the FM9001 microprocessor does not include a memory; some of the signals are used to interface to an external memory. Likewise, there are signals devoted to testing the FM9001. Not all signals are relevant to all the levels of formal specification. For example, the power and ground pins VSS and VDD never appear in *any* of our specifications; we always assume that the power is turned on! All of the other signals appear in our netlist-level specification, although the clock input CLK is given special treatment. The high-level specification of the FM9001 does not include any signals, it is a self-contained specification that includes the memory.

Figure 8 is a drawing of the physical pinout for the FM9001. The physical testing mentioned above was performed by physically connecting the named pins to our test equipment.

The adequacy of the behavioral formal description of the FM9001, via the function `FM9001`, is demonstrated by the fact that it has been possible to verify the correctness of a compiler targeting the FM9001, for the programming language Piton[31, 32, 33], and to prove the correctness

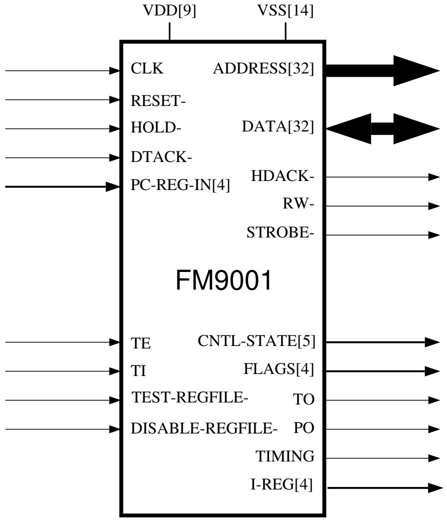
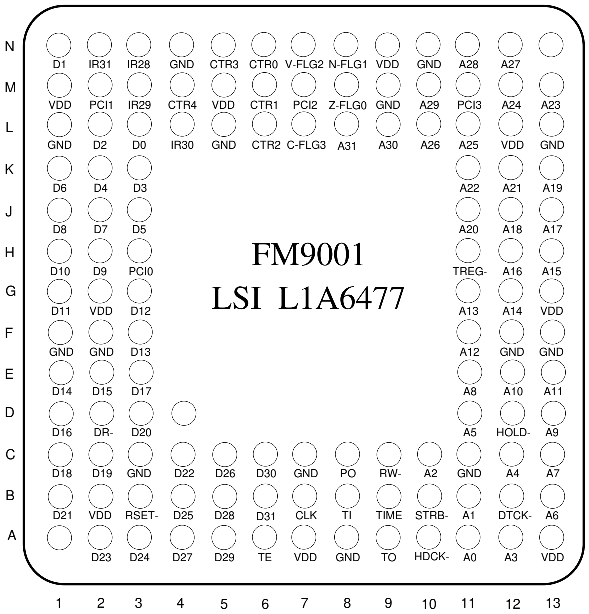


Fig. 7. Signal Groups of the FM9001



A:	Addr-out	HDCK:	HDack–
CTR:	Ctrl-state	IR:	I-reg
D:	Data-bus	PCI:	PC-reg-in
DR:	Disable-regfile–	RSET:	Reset–
DTCK:	Dtack–	STRB:	Strobe–
FLG:	Flags	TIME:	Timing
GND:	VSS	TREG:	Test-regfile–

Fig. 8. Pinout of the FM9001

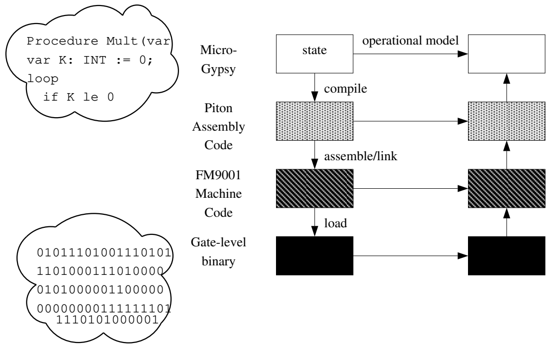


Fig. 9. The CLI Hardware-Software Stack

of a number of software applications written in Piton and in FM9001 object code. These verified programs include a Nim-playing program[37] and a real-time button/light system [38]. For a diagram briefly summarizing this edifice of hardware and software, see Figure 9.

`DUAL-EVAL` is not a commercial hardware description language. To make use of circuits specified and verified with `DUAL-EVAL`, we translate `DUAL-EVAL` into other hardware description languages. In the case of the FM9001, we translated `DUAL-EVAL` into LSI Logic's NDL for fabrication purposes. (Our program that prints `DUAL-EVAL` netlists as NDL is only a few lines long.) We have also translated the FM9001 into VHDL. We expect that would not be too difficult to translate `DUAL-EVAL` netlists into other hierarchical hardware description languages.

X. Testing the FM9001

After being fabricated as an LSI Logic ASIC, the FM9001 was extensively tested both with user software and with a Tektronix chip tester. For over 1,000,000 vectors, we compared the behavior of a sample FM9001 on the Tektronix tester against the next state and outputs predicted by `DUAL-EVAL` and were pleased that the chip behaved as predicted. For extensive details on the testing, see the report "Testing the FM9001 Microprocessor" [1].

One purpose of testing the fabricated devices, of which we received fifty, was to insure that these integrated circuits were free of physical defects. Testing is necessary because even given a logically perfect netlist, one may encounter problems with the physical manufacturing process, e.g., dirt, broken wires, and cracked packages.

A second reason for testing was to validate the assumptions that we made about the LSI Logic primitives when we formalized the behavior of the primitive elements used in making the FM9001. Our lowest level formal model was one that defined such LSI Logic primitives as an AND gate and a one-bit latch. Even though LSI Logic provides a databook [28] of gate-level primitives, their actual gate array devices are composed of columns of N-type and P-type transistors. It was important to us to discover whether the translation process from our gate-level model to LSI Logic's Network Description Language (NDL) and then on to transistors all worked as we expected.

Finally, we wanted to validate our effort in producing a verified hardware and software computing platform. The CLI "short stack" [2] includes the FM9001 microprocessor as a base, upon which the Piton assembler [31, 32], the Micro-Gypsy compiler [41, 42, 43], and the Micro-Nqthm compiler [18] have been proven to operate correctly. We find completion of this verification exercise to be compelling evidence [29] that we do not have to settle for hardware and software validated only with the conventional test-oriented approaches.

We also note that Bhaskar Bose [4] has formally derived an equivalent piece of hardware (at the instruction level) using the Digital Design Derivation (DDD), system starting from the FM9001 behavioral specification. The synthesized microprocessor was implemented using FPGA technology.

XI. Conclusion

We have given in its entirety a formal hardware description language, `DUAL-EVAL`. The `DUAL-EVAL` language allows the definition of Mealy machines and their hierarchical composition. We have described a general-purpose, 32 bit microprocessor, the FM9001, which has been completely specified as a `DUAL-EVAL` netlist. The implementation of the FM9001 is described as a `DUAL-EVAL` netlist containing a reference for every primitive gate, latch, register, I/O buffer, and wire that is required for the FM9001 implementation. Both the syntax and the semantics of this netlist have been mechanically checked: the netlist is well-formed and it has been proven to implement the FM9001 specification. We have described a mechanically checked proof that this FM9001 netlist meets a behavioral specification. The FM9001 has been successfully fabricated as an LSI Logic CMOS ASIC and extensively tested.

References

[1] Kenneth L. Albin, Bishop C. Brock, Warren A. Hunt, Jr., and Lawrence M. Smith. Testing the FM9001 Microprocessor. Technical Report 90, Computational Logic, Inc., 1717 W. Sixth St., Suite 290, Austin, Texas 78703, 1994.

[2] William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, and William D. Young. An Approach to Systems Verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[3] 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.

[4] Bhaskar Bose. DDD-FM9001: Derivation of a Verified Microprocessor. PhD Dissertation, Indiana University, December 1994.

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

[6] R.S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988. For prover sources, documentation of the logic, and examples, see ftp://ftp.cli.com/pub/nqthm/nqthm-1992/nqthm-1992.tar.Z or ftp://ftp.cs.utexas.edu/pub/boyer/nqthm-1992.tar.Z.

[7] Bishop Brock. Transforming Functional Hardware Models into Structural Models. Internal Note 150, Computational Logic, Inc., July 1989.

[8] Bishop C. Brock and Jr. 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.

[9] Bishop C. Brock, Warren A. Hunt, Jr., and William D. Young. Introduction to a Formally Defined Hardware Description Language. *Theorem Provers in Circuit Design*, V. Stavridou, T. Melham, and R. Boute, eds., North-Holland, pp. 3–35, 1992.

[10] Bishop C. Brock, Warren A. Hunt, Jr., and Matt Kaufmann. The FM9001 Microprocessor Proof. Technical Report 86, Computational Logic, December, 1994.

[11] 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.

[12] 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.

[13] A. Camilleri, M. Gordon, and T. Melham. Hardware Verification Using Higher-order Logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 43–67. Elsevier Science Publishers, Amsterdam, 1987.

[14] 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.

[15] 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.

[16] Ivan V. Filippenko. VHDL Verification in the State Delta Verification System. In *1991 ACM SIGDA International Workshop on Formal Methods in VLSI Design*, January 1991.

[17] 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.

[18] Arthur D. Flatau. *A Verified Implementation of an Applicative language with Dynamic Storage Allocation.* Ph. D. Dissertation, The University of Texas at Austin, December, 1992. Also available (minus certain appendices) from Computational Logic as CLI Technical Report 83.

[19] M. J. C. Gordon and T. F. Melham (editors). *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.

[20] 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.

[21] M.J.C. Gordon. The Semantic Challenge of Verilog HDL. Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science (LICS'95).

[22] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor.* LNCS 795, Springer-Verlag, 1994.

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

[24] Warren A. Hunt, Jr. and Bishop Brock. A Formal HDL and Its Use in the FM9001 Verification. In C.A.R. Hoare and M.J.C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, pages 35–48. Prentice-Hall International Series in Computer Science, Englewood Cliffs, N.J., 1992.

[25] Graham Birtwistle, Jeffery Joyce, and Mike Gordon. Proving a Computer Correct in Higher Order Logic. Technical report, University of Calgary, Department of Computer Science, August 1985.

[26] 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.

[27] C. Delgado Koos and P. T. Breuer (editors). *Formal Semantics for VHDL.* Kluwer, 1995.

[28] LSI LOGIC. 1.5-Micron Array-Based Products Databook. LSI Logic Corporation, Milpitas, CA. 1990.

[29] John McCarthy. It's Past Time for Practical Computer Checked Proofs of Program Correctness. *Computational Logic*, Symposium Proceedings, Brussels, Esprit Basic Research Series, DG XIII, Commission of the European Communities, Springer-Verlag, November 1990.

[30] Kenneth L. McMillan. *Symbolic Model Checking* Kluwer, 1993.

[31] J Strother Moore. PITON: A Verified Assembly Level Language. Technical Report 22, Computational Logic, 1988. To appear as a book in the Kluwer's series on automated reasoning under the title *Piton: A Mechanically Verified Assembly-Level Language.*

[32] J Strother Moore. A Mechanically Verified Language Implementation. *Journal of Automated Reasoning*, 5(4):493–518, December 1989. Also published as CLI Technical Report 30.

[33] J Strother Moore, et al. Special Issue on System Verification. *Journal of Automated Reasoning*, Vol. 5, No. 4, pp. 409-530, 1989.

[34] Carl-Johan H. Seger and Randy E. Bryant. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. Technical Report 93-8. Computer Science Department, University of British Columbia, 1993.

[35] Mandayam Srivas and Mark Bickford. Formal Verification of a Pipelined Microprocessor. *IEEE Software*, 7(5):52–64, September 1990.

[36] Mandayam Srivas and Steven P. Miller. Formal Verification of a Commercial Microprocessor. Technical Report SRI-CSL-95-4. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1995.

[37] Matthew Wilding. A Mechanically Verified Application for a Mechanically Verified Environment. *Fifth Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, LNCS 697, pp. 268–279, Springer-Verlag, 1993. Also available as Technical Report 78, Computational Logic, 1994.

[38] Matthew Wilding. Specification, Construction, and Mechanically-checked Verification of a Simple Real-time Executive. Computational Logic Technical Report (in preparation), 1995.

[39] Phillip J. Windley. Formal Modeling and Verification of Microprocessors. *IEEE Transactions on Computers*, Vol. 44, No. 1, Jan. 1995, pp. 54–72.

[40] Michael Yoeli, editor. *Formal Verification of Hardware Design*. IEEE Computer Society Press Tutorial, 1990.

[41] William D. Young. *A Verified Code Generator*. Ph. D. Dissertation, University of Texas at Austin, 1988. See also Computational Logic Technical Report Number 37, 1988.

[42] William D. Young. A Mechanically Verified Code Generator. *Journal of Automated Reasoning*, Vol. 5, Number 4, (December, 1989), pp. 493–518. Also available from Computational Logic as Technical Report 37.

[43] William D. Young. Verified Compilation in Micro-Gypsy. *Proceedings of the Software Testing, Analysis and Verification Symposium*, Key West Florida, December, 1989, pp. 20–26.