N94-18372

A Verification Logic Representation of Indeterministic Signal States

J. W. Gambles and P. J. Windley
NASA Space Engineering Research Center for VLSI Systems Design
University of Idaho
Moscow, Idaho 83843

Abstract - **The integration of modern CAD tools with formal verification environments require translation from hardware description language to verification logic. A signal representation including both unknown state and a degree of strength indeterminacy is essential for the correct modeling of many VLSI circuit designs. A higher-order logic theory of indeterministic logic signals is presented.**

1 Introduction

As higher transistor counts increase the complexity of VLSI circuits and the number of potential test cases explode, formal verification methods promise value in design fault exclusion. Before verification is accepted by design engineers, stand alone verification tools that are used in the academic research arena must be integrated with the CAD tools being used by VLSI designers. One major benefit of this integration is that VLSI designers will enjoy increased confidence that abstract behavioral models are correct. There are several reasons a VLSI designer may choose to use abstract behavioral models. In a top-down design, a behavioral description may be used to simplify circuit understanding before the implementation is designed. A behavioral model can be utilized as part of a simulation of the entire system at an early date. After the circuit structure is designed and modeled, the logic simulation of complex systems can become very slow. The simulation can be made faster by replacing circuit blocks with the corresponding behavioral model. The problem with these design approaches is that there is currently no way to relate the circuit structural model to the abstract behavioral model. Having a verification tool available in the VLSI CAD tool suite would allow these models to be related through mathematical analysis.

The hardware description languages (HDL) used by VLSI CAD tools can provide the link between these tools and the verification environment. Engineers can design using the CAD tool HDL and this description can be automatically translated for use in the verification tool. This paper examines the translation of logic signal representations from the BOLT (Block Oriented Logic Translator) HDL, used in the NOVA simulation engine, to the HOL theorem proving system.

2 HOL

HOL is a general theorem proving system developed at the University of Cambridge [4,6] based on Church's theory of simple types, or higher-order logic. Higher-order logic is suitable for specifying all aspects of hardware, including both structure and behavior [6,8]. In using higher-order logic, predicates are defined to represent both circuit primitives and behavioral definitions [4]. First-order logic is well suited to represent simple combinational circuits, but not sequential circuits. In higher-order logic, variables are allowed to range over functions and predicates which make it suitable for representing sequential circuit behavior [8]. HOL is not an automated theorem prover but is more than simply a proof checker, falling somewhere between these two extremes. Translation from BOLT descriptions to HOL predicates requires that HOL primitives be defined to correspond to the BOLT circuit representations.

Symbols in HOL are represented by strings of ASCII characters. Conjunction, disjunction, negation, implication, and equality are represented by `/\`, `\/`, `~`, `==>`, and `=` respectively. Universal quantification (for all) is symbolized `!` and existential quantification (there exists) is `?`. The function composition operator is `o` and the conditional expression "if *a* then *b* else *c*" is symbolized `a => b | c`.

3 Logic States and Strengths

Few modern VLSI circuits are designed using only classical logic gates [3,10]. In designs using pass-transistor, tri-state, and pre-charge logic, it is common for circuit nodes to be driven from multiple circuit elements. These multiple drivers are designed to have differing drive strengths in order for one to dominate over another in cases of contention. The drive strength can be considered to be closely related to current drive (charge sourcing) capability [7,2]. The signal values represented in the NOVA simulation engine are an extension of Bryant's lattice theoretic approach [7,11]. In the lattice theoretic approach the elements in the domain of signal values represent the combination of logic state, from the set **True**, **False**, and **Unknown**; and a signal strength. These signal values form a partially ordered set with their order based on strength dominance when circuit output values are combined.

While Bryant later abandoned the lattice theoretic approach [2] stating "while this approach at first seems very elegant, it cannot adequately describe the effects of transistors in the X (**Unknown**) state, " Cameron and Shovic have shown that the problem with the **Unknown** state can be corrected by extending the domain of signal values to include some degree of strength indeterminacy [3]. Thus, the signal values are extended to represent both logic states and a *range* of signal strength.

The **Unknown** state can be the result of a node connected to two drivers, one driving to a **True** and the other driving to a **False**, neither driver having sufficient strength to dominate the other; or simply a node whose voltage is not yet known. Combining the cases of "invalid" logic level and "valid but not known" into a single **Unknown** state simplifies the simulation algorithm but may make the simulator pessimistic since it will propagate the Unknown state when resolving some circuit nodes[2].

We refer to the combination of state and strength information as STATES. The STATES representation presented here is consistent with that presented in [3,10] except the total number of strengths N, is extended to include a weakest strength, Nil, which represents a node that is disconnected from all charge sources. By definition, a signal being driven by the Nil strength must be at the Unknown state.

3.1 Representation of STATES

Given the set of states True, False, and Unknown and a fully ordered set of strengths $\sigma_1, \sigma_2, \ldots$, and σ_N we can define STATES. The STATES corresponding to the states True and False are represented as a triple **Kbd** where:

K is 1 or 0 representing the logic state True or False;

bd represents a indeterminate *range* of strengths where:

- **b** is the strongest possible strength ($\sigma_1 \leq \mathbf{b} \leq \sigma_{N-1}$) which sets a lower bound on the strength of a signal that can overdrive this state;
- **d** is the weakest possible strength ($\mathbf{b} \leq \mathbf{d} \leq \sigma_{N-1}$) which sets a upper bound on the strength of a signal that this state can overdrive.

The STATES corresponding to the Unknown state are represented as a triple **Xpq** where:

X represents the Unknown state;

p is the strongest possible strength driving toward 0 ($\sigma_1 \leq \mathbf{p} \leq \sigma_{N-1}$) which sets a lower bound on the strength of a signal that can overdrive this state to a 1;

q is the strongest possible strength driving toward 1 ($\sigma_1 \leq \mathbf{q} \leq \sigma_{N-1}$) which sets a lower bound on the strength of a signal that can overdrive this state to a 0.

3.2 The Number of STATES

For N strengths the number of True and False STATES is:

$$TF\_STATES(N) = 2((N-1) + (N-2) + \ldots + 1) = (N-1)(N) \tag{1}$$

For the Unknown state the number of STATES is:

$$X\_STATES(N) = (N-1)^2 + 1 \tag{2}$$

The plus one term in equation (2) represents the combination of Unknown state and weakest strength, $\sigma_N = Nil$. This STATE is referred to as Nil. Thus, the total number of STATES for N strengths is equal to:

$$TOTAL\_STATES(N) = 2N^2 - 3N + 2 \tag{3}$$

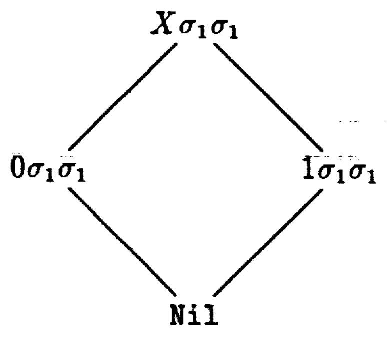

Figure 1: Base Case Signal Lattice (N=2)

4 STATES Theory

A complicated algorithm for determining the result of combining STATES is presented in [3]. This algorithm is not satisfactory for use in HOL. We have developed a lattice that describes the result of joining two signals. In this lattice theoretic approach to signal strengths, the *join* (least upper bound) operation represents the resolution of contending circuit elements [11].

The lattice structure is described through the notion of immediate superiors or *covers*. For two elements, a and b of a partially ordered set, *a covers b* if and only if $a > b$ and there exists no element x of the partially ordered set such that $a > x > b$ [1]. A list of all of the elements and covers completely describe a lattice. The covers can also be used to define a graph of the lattice. The vertices of the graph are the elements and the segments of the graph represent the covers. If the graph is drawn such that whenever x covers y, the vertex x is *higher* than the vertex y, it is called a "Hasse diagram" of the lattice [1].

4.1 Defining STATES Lattice Structure

Given the base case $N = 2$ ($N = 1$ is a trivial case of one single `STATE`, `Nil`) there are four `STATES` and no strength indeterminacy, meaning there is only a single value (σ_1) within the range of possible strengths. There are four covers and the lattice Hasse diagram is as presented in [7,11], a simple diamond (Figure 1).

To extend a N strength Hasse diagram (lattice) to $N + 1$ strengths:

1. Add three `STATES` and four covers to form a new diamond at the bottom of the N strength diagram by replacing `Nil` with $X\sigma_N\sigma_N$, adding $0\sigma_N\sigma_N$ and $1\sigma_N\sigma_N$ each covered by $X\sigma_N\sigma_N$ and placing `Nil` at the bottom of the diagram covered by both $0\sigma_N\sigma_N$ and $1\sigma_N\sigma_N$.

2. For each $M = N$ to 2, by -1, add the following `STATES` and covers:

(a) $X\sigma_{M-1}\sigma_N$ covered by $0\sigma_{M-1}\sigma_{N-1}$ and covering $X\sigma_M\sigma_N$

(b) $X\sigma_N\sigma_{M-1}$ covered by $1\sigma_{M-1}\sigma_{N-1}$ and covering $X\sigma_M\sigma_N$

(c) $0\sigma_{M-1}\sigma_N$ covered by $X\sigma_{M-1}\sigma_N$ and covering $0\sigma_M\sigma_N$

(d) $1\sigma_{M-1}\sigma_N$ covered by $X\sigma_N\sigma_{M-1}$ and covering $1\sigma_M\sigma_N$

4.2 The Number of Covers

The total number of covers for N strengths is equal to:

$$COVERS(N) = 4N^2 - 10N + 8 \tag{4}$$

4.3 The Lattice Structure for NOVA

The NOVA simulation engine and BOLT HDL have been selected for this research so that we may have access to commercial-scale designs written by nonacademic VLSI designers while a translation tool to HOL is developed. In NOVA, $N = 4$ and $\sigma_1 = a$ (active), $\sigma_2 = r$ (resistive), $\sigma_3 = f$ (float) and $\sigma_4 =$ Nil. Note that float > Nil and can be used to represent signal levels at charged capacitive nodes. For $N = 4$, equation (3) yields 22 `STATES` and equation (4) yields 32 covers. The Hasse diagram for the `STATES` and covers for NOVA is shown in figure 2. In addition to identifying the list of covers required to define the lattice structure in the verification logic, the Hasse diagram also provides a quick, visual understanding of the resolution of joined `STATES`.

5 Implementing STATES in HOL

The HOL system includes a type definition package that allows the user to define new types and prove theorems about essential properties of the new type. The type package automatically carries out much of the necessary formal proof required for a new type definition. Theorems about the new type are proven, rather than simply postulating axioms for the new type, in order to avoid the introduction of inconsistency into the logic [9]. A new type for signal values, called `strength`, is defined in HOL by enumeration of all of the `STATES`. Properties proven about the new type include each value being distinct, an induction theorem, and a case analysis (perfect induction) theorem. The STATES lattice is defined by enumeration of the covers and the function `join` is defined to be the least upper bound. Once the `join` function definition is complete, consistency of proofs that utilize `join` are insured by formal proof of the lattice theoretic obligations [11] for the join operation. These obligations are:

1. Idempotence. For all a `STATES`, `join` $a\ a = a$.

2. Commutativity. For all a and b `STATES`, `join` $a\ b =$ `join` $b\ a$.

3. Associativity. For all a, b and c `STATES`, `join` a (`join` $b\ c$) $=$ `join` (`join` $a\ b$) c.

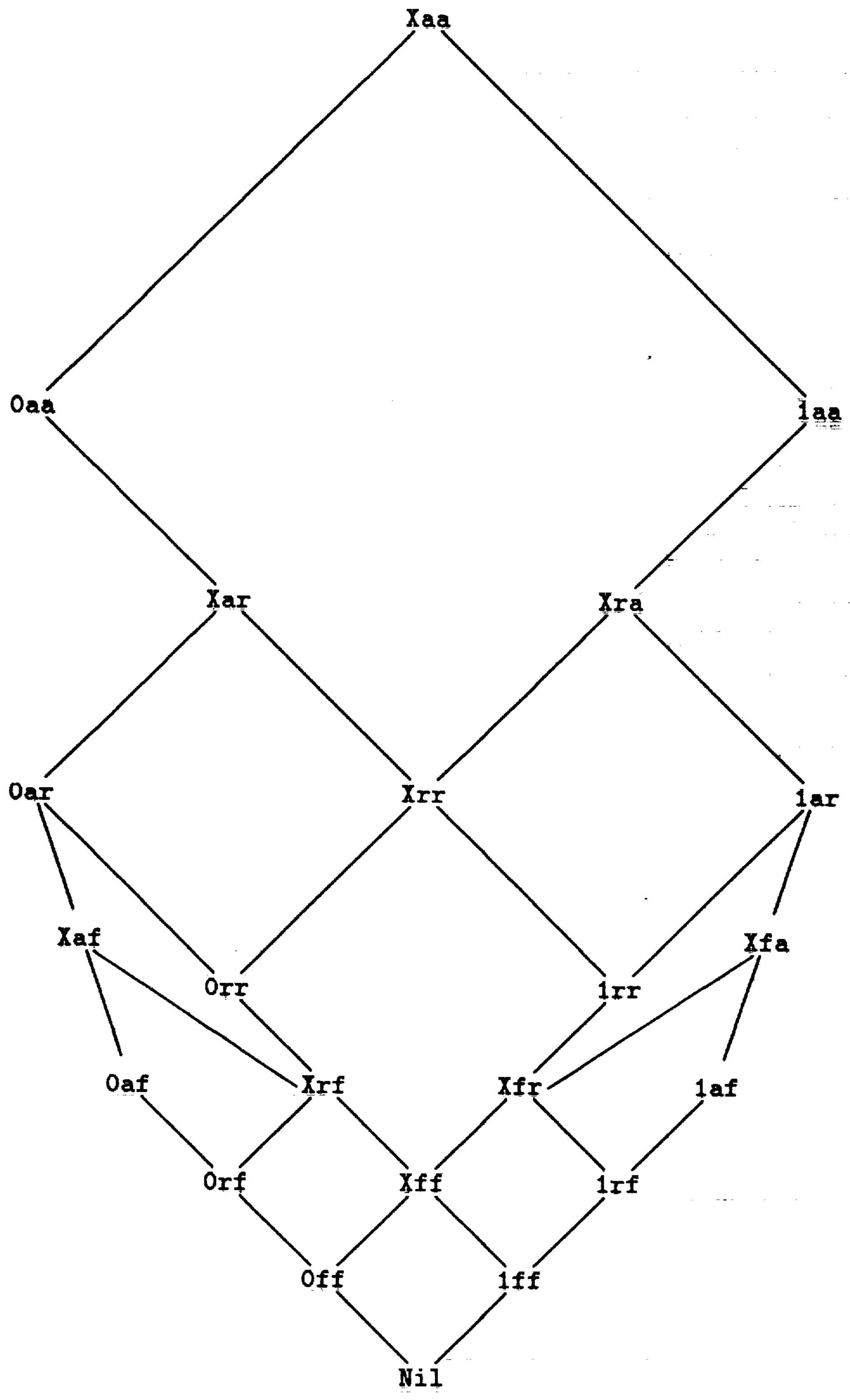

Figure 2: Signal Lattice for N=4 (NOVA)

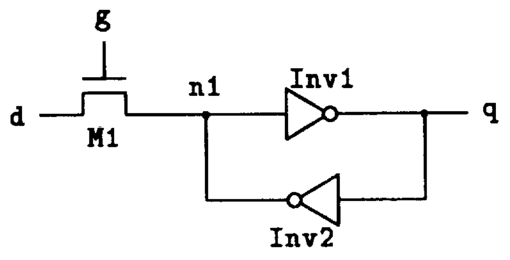

Figure 3: Memory Cell Schematic Diagram

4. Existence of bottom. For all *a* `STATES`, `join` *a* `Nil` = *a*.

5.1 STATES Abstraction Function

Typically a behavioral specification is defined in terms of boolean values. An abstraction function is required to relate `STATES`, used in structural specifications, to boolean values.

```
STATES_ABS sig = ((sig=1aa)\/(sig=1ar)\/(sig=1rr)\/
                  (sig=1af)\/(sig=1rf)\/(sig=1ff)) => T |
                 ((sig=0aa)\/(sig=0ar)\/(sig=0rr)\/
                  (sig=0af)\/(sig=0rf)\/(sig=0ff)) => F |
                                                      ARB
```

The `Unknown` `STATES` are assigned a value `ARB`, defined to be an arbitrarily chosen boolean value.

6 Theory Demonstration

A static memory circuit cell, implemented with gate level and pass transistor primitives, is used to demonstrate the `STATES` theory (Figure 3). Without a signal value representation that realizes output dominance this circuit cannot be correctly modeled. Fundamental to the operation of this circuit is that the output strength of pass-transistor `M1` dominates the output of inverter `Inv2` to force node `n1` to the state of the input `d` while the gate `g` is `True` (high voltage). The feedback inverter `Inv2` acts to store the state, by dominating the pass-transistor after the gate goes `False`, turning the transistor off.

6.1 The Circuit Primitives

The memory cell structure includes three predicate definitions; a pass-transistor element, inverter elements, and the `JOIN` operation. Time is represented as a number (`num`) stream and circuit signals are defined to be functions of type `num` to type `strength`.

The behavioral model of the cell is not defined for the gate input being at an unknown state. A simplified pass-transistor model is used that defines that the signal at the drain is equal to the signal at the source if the gate is **True**, else it is **Nil**.

```
NTRAN (g,s,d) =
 ! t.
   d t = ((g t =1aa)\/(g t =1ar)\/
          (g t =1rr)\/(g t =1af)\/
          (g t =1rf)\/(g t =1ff)) => s t |
                                     Nil
```

The inverter predicate has five arguments. The first three arguments are of type **strength** and define the possible inverter output STATES. The first is the output **STATES** for a **True** output, the second for a **False** output, and the third the **Unknown** state output. The **Unknown** output value is derived from the strongest **True** and **False** strengths. The fourth and fifth arguments are signal functions of type **num** to type **strength**. The fourth is the inverter input and the fifth is the output.

```
INV 1s 0s Xs (in,out) =
 ! t.
  out t =(((in t =1aa)\/(in t =1ar)\/
           (in t =1rr)\/(in t =1af)\/
           (in t =1rf)\/(in t =1ff)) => 0s |
          ((in t =0aa)\/(in t =0ar)\/
           (in t =0rr)\/(in t =0af)\/
           (in t =0rf)\/(in t =0ff)) => 1s |
                                        Xs )
```

6.2 JOIN

The **JOIN** predicate performs two operations. It determines the resulting signal value of combining circuit outputs by applying the **join** function. The second operation is related to the sequential behavior of a charge storage node. The capacitance of a node may result in a time delay when the node is driven to a new signal level. The delay time increases as the strength of the driving signal decreases. This sequential behavior is modeled as having a variable delay, whose length is based on the strength of the **join** function result. [5,7]. The Hasse diagram shows the relative strength of **STATES** and can be used to abstract the delay values for individual **STATES** by segregating them into horizontal bands on the diagram. All **STATES** within a common band have the same delay and the delay is longer for lower bands. For cases where it is desired to model different delays for rise and fall times the diagram can be segregated right from left also.

The demonstration cell is modeled as having two possible delays. When the pass-transistor is turned on, the storage node at the join is driven by an active strength and the delay is defined to be zero. When the pass-transistor is turned off, the storage node is driven by the resistive strength of the feed-back inverter and the delay is defined to be one.

```
JOIN (s',s'',s:num->strength) =
 ! t. let sig = join (s' t) (s'' t) in
      ((sig = 0aa) \/
       (sig = 1aa) \/
       (sig = Xaa) \/
       (sig = Xar) \/
       (sig = Xra)) => (s t = sig)       |
                       (s (t+1) = sig)
```

6.3 The Structural Description

A BOLT description of the cell is:

```
MODULE Q .CELL G D;

BEGIN
   N1  .NTRAN G D;
   Q   .INVR  N1;
   N1  .INVR  Q  (STR='RR');
END;
```

The STR='RR' parameter in the second INVR invocation defines the output strength of that inverter as resistive. The default value used for the first invocation is active. The HOL structural specification of the cell is:

```
cell_IMP (d,g,q) =
 ? n1 n1' n1'':num->strength .
   NTRAN (g,d,n1')             /\
   INV 1aa 0aa Xaa (n1,q)      /\
   INV 1rr 0rr Xrr (q,n1'')    /\
   JOIN (n1',n1'',n1)
```

6.4 The Behavioral Description

When the gate of the pass-transistor is True the cell is *writing* the input and the output, q, follows as the inverse of d. When the gate is False the cell is *storing* the previous data. The HOL behavioral description is:

```
cell_SPEC (d,g,q) =
 !t.
  (g t) => (q t = ~d t)    |
            (q (t+1) = q t)
```

6.5 The Cell Verification

Because the operation of the cell requires that the output of the pass-transistor dominate the resistive strength output of INV2 and the pass-transistor is not an amplifier, there is a validity condition that the signal applied to input d must be stronger than resistive. This condition is required for proper circuit operation and is not simply a verification artifact.

```
   Valid1 (d) =
   ! t.
   (d t = 1aa ) \/ (d t = 0aa)
```

Because the behavior of the cell is defined only for boolean value signals at the gate, there is a validity condition for the gate that it be either a **True** or **False** state. This condition yields a 12 way case analysis in the proof, but is easily reduced to needing to consider only the two cases of writing and storing.

```
Valid2 (g) =
 ! t.
   (g t = 1aa) \/ (g t = 1ar) \/ (g t = 1rr) \/
   (g t = 1af) \/ (g t = 1rf) \/ (g t = 1ff) \/
   (g t = 0aa) \/ (g t = 0ar) \/ (g t = 0rr) \/
   (g t = 0af) \/ (g t = 0rf) \/ (g t = 0ff)
```

The verification of the cell entails proving that the cell structural description and validity conditions logically imply the behavioral specification. The theorem proven is:

```
|- (Valid1 (d) /\ Valid2 (g) /\ cell_IMP(d,g,q)) ==>
    cell_SPEC(STATES_ABS o d, STATES_ABS o g,STATES_ABS o q)
```

7 Future Work

The theory of signal lattices presented in this paper is an important first step in linking BOLT and HOL. Future steps include:

1. Developing and validating a set of HOL theories corresponding to the primitive components in the NOVA library.
2. Writing a formal semantics for BOLT.
3. Embedding BOLT's formal semantics in HOL.

These steps do not include work on translating NOVA behavioral models to HOL, a difficult, but necessary task.

8 Conclusion

The first step in the integration of CAD VLSI design tools with a verification tool is the translation of the HDL representations into the verification logic. A verification logic theory has been presented for reasoning about an indeterministic signal value representation based on a lattice approach. This work is necessary because the previous algorithm for joining indeterministic signal values is not suitable for a verification logic environment. The suitability of the lattice approach is demonstrated through the verification of a static memory cell. The lattice diagram presented also quickly provides to users the result of combining different valued indeterminate signals.

9 Acknowledgements

This research was supported by NASA under Space Engineering Research grant NAGW-1406.

References

[1] Birkhoff, G., *Lattice Theory,* American Mathematical Society, 1948.

[2] Bryant, R. E., "A Switch-Level Model and Simulator for MOS Digital Systems," IEEE Transactions on Computers, Vol. C-33, pp.160-177, February 1984.

[3] Cameron, K. B. and Shovic, J. C., "Calculating Minimum Logic State Requirements for Multi-Strength Multi-Value MOS Logic Simulators, " 1987 IEEE International Conference on Computer Design: VLSI in Computers & Processors, IEEE Computer Society Press, pp. 672-675, 1987.

[4] Camilleri, A., Gordon, M. and Melham, T., "Hardware Verification Using Higher Order Logic," in D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs,* Elsevier Scientific Publishers, pp. 43-67, 1987. Also Technical Report No. 91, University of Cambridge Computer Laboratory, September 1986.

[5] Dhingra, I. S., "Formal Validation of An Integrated Circuit Design Style," in G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis,* Kluwer Academic Publishers, pp. 293-321, 1988. Also Technical Report No. 115, University of Cambridge Computer Laboratory, August, 1987.

[6] Gordon, M. J. C., "HOL: A Proof Generating System for Higher-Order Logic," in G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis,* Kluwer Academic Publishers, pp. 73-128, 1988. Also Technical Report No. 103, University of Cambridge Computer Laboratory, August, 1987.

[7] Hayes, J. P., "A Unified Switching Theory with Applications to VLSI Design," Proceedings of the IEEE, Vol. 70, No. 10, pp.1140-1151, October 1982.

[8] Melham, T. F., "Abstraction Mechanisms for Hardware Verification," in G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis,* Kluwer Academic Publishers, pp. 267-291, 1988. Also Technical Report No. 106, University of Cambridge Computer Laboratory, May, 1987.

[9] Melham, T. F., "Using Recursive Types to Reason About Hardware Verification," Technical Report No. 135, University of Cambridge Computer Laboratory, May, 1988.

[10] Miles, L., Prins, P., Cameron, K., and Shovic, J., "NOVA: A New Multi-Level Logic Simulator," 2nd NASA SERC Symposium on VLSI Design, pp. 4.1.1-4.1.13, 1990.

[11] Ullman, J. D., *Computational Aspects of VLSI,* Computer Science Press, 1984.

Formal Verification of State Machines

M. Alahmad and P. Windley
NASA Space Engineering Research Center
for VLSI System Design
University of Idaho
Moscow, Idaho 83843

Abstract – **A formal specification of VLSI state machines based on a sequence invariant architecture is presented. The behavioral description represents a logical description of any synchronous state machine. The structural specification represents an adoptive architecture developed using VLSI technology to implement the state machine. This specification becomes a tool for future verification and specification of state machines using dedicated machines and/or alternative technologies. The verification of the state machine is done in HOL, a theorem proving system. Using HOL, the verification shows analytically that the circuit structure has the desired behavior.**

1 Introduction

With the advancement of integrated circuit technology, the need for new methods of ensuring design correctness is becoming more prominent. Simulation remains the dominant method in use, but, recently, interest has grown in using formal logical analysis to show the correctness of digital systems.

Formal verification of hardware involves using theorem-proving techniques to verify that a stated behavioral definition of a circuit is a logical consequence of the structural description of the circuit, i.e., proving that the structure of the circuit forces it to behave as stated. This paper presents a formal specification and verification of a general state machine. The specification describes the behavior and structure of the state machine. The behavioral specification is a logical representation of a state machine. Using a particular design in VLSI technology, a structural description based on the Sequence Invariant Architecture is described. The structure clearly specifies how components are connected and built to achieve the operation of the state machine. The verification shows, by analysis, that the structural specification implies the behavioral specification using a theorem proving system known as HOL [1]. Hence, the VLSI architecture is capable of implementing any state machine.

2 The HOL System

As described by Birtwistle and Subrahmanyam [3], the HOL system ('HOL' standing for 'higher order logic') is designed to facilitate the interactive generation of formal proofs. A logic in which problems can be expressed is interfaced to a *programming language* in which proof procedures and strategies can be encoded. The combination enables deduction in logic (in the sense of chains of primitive inference steps) to be produced by invocation of programming constructs at a higher level of abstractness.

The logic part of HOL is conventional higher-order logic. New types, constants and axioms can be introduced by the user, and organized in logic *theories*. The programming language of HOL is ML (for 'meta-language'). The *type discipline* of ML ensures that the only way to create *theorems* in the object logic is by performing proofs; theorems have the ML type *thm*, objects of which can only be constructed by the application of interface rules to other theorems or axioms.

3 Sequential Circuits Overview

Sequential circuits are categorized as either synchronous or asynchronous, depending upon whether or not the behavior of the circuit is clocked at discrete instants of times. The operation of synchronous sequential circuits (the topic of this paper) is controlled by a synchronizing pulse signal called a clock pulse or simply a clock.

Sequential machines are usually represented by state diagrams or state tables (flow tables). A flow table has a row corresponding to every internal state of the machine and a column corresponding to every possible input. The entry in row q_i and column I_m represents the next state produced if I_m is applied when the machine is in state q_i. Table 1 shows a flow table for an arbitrary circuit with six-states and three inputs. Once the flow table is constructed for a given circuit, a state assignment is performed. A state assignment is the encoding of the states of the flow table with the internal state variables $(y_1, y_2, ..., y_n)$. Table 2 shows the state assignment and the next state entries assignment for Table 1. Finally, the next state equations are derived from the state assignment using Karnaugh map techniques. We can also derive an equation that describes the output behavior from the flow table.

3.1 SISM Overview

An adaptive hardware architecture has been developed [2], that enables the designer to design any sequential circuit based on the width of the machine w, and the number of control inputs I, without a knowledge about the sequence to be incorporated. This adaptive architecture is called a Sequence Invariant State Machine (SISM) design.

With the SISM realization, any flow table can be implemented without a change in the hardware configuration. That is given w, and I, a hardware circuit is easily derived, that can implement any state machine that has a maximum of I control inputs, and 2^w internal states.

3.2 Architecture And Operation

	I_1	I_2	I_3
A	C, 1	B, 1	A, 0
B	D, 0	C, 1	B, 0
C	E, 0	D, 0	C, 0
D	F, 1	E, 1	D, 1
E	A, 0	F, 0	E, 1
F	B, 0	A, 1	F, 1

Table 1: General 6-states, 3-input flow table.

y_1	y_2	y_3		I_1			I_2			I_3		
0	0	0	A	0	1	0, 1	0	0	1, 1	0	0	0, 0
0	0	1	B	0	1	1, 0	0	1	0, 1	0	0	1, 0
0	1	0	C	1	0	0, 0	0	1	1, 0	0	1	0, 0
0	1	1	D	1	0	1, 1	1	0	0, 1	0	1	1, 1
1	0	0	E	0	0	0, 0	1	0	1, 0	1	0	0, 1
1	0	1	F	0	0	1, 0	0	0	0, 1	1	0	1, 1
1	1	0	G	0	0	0, 0	0	0	0, 0	0	0	0, 0
1	1	1	H	0	0	0, 0	0	0	0, 0	0	0	0, 0

Table 2: State Assignment for Table 1.

Figure 1 shows a general SISM architecture, this architecture can be used to implement one of the next state variables in Table 2.

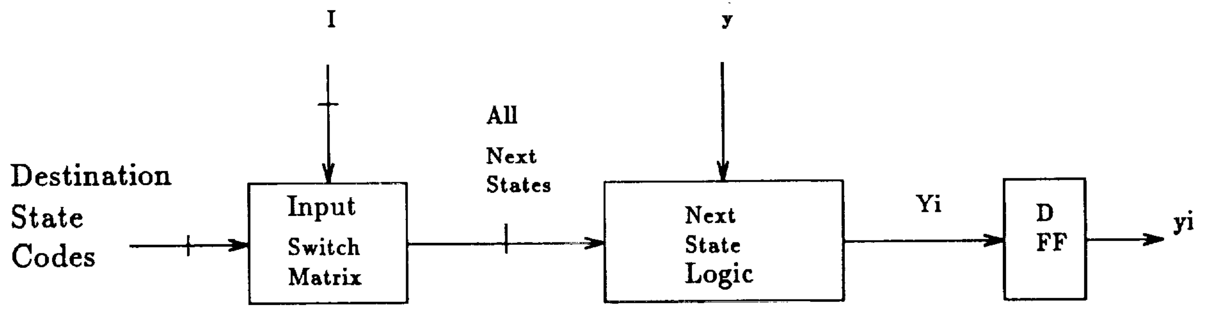

Figure 1: General SISM Architecture.

The architecture contains the following components:

- The destination state codes are derived from the next state entries in the state assignment table by inspection. For example, the destination state codes for state B and state variable y_i are the next state bits Y_i associated with state B. Therefore, the destination state codes for state B are (000, 110, 101) under control inputs $(I_1; I_2; I_3)$ and variables $(y_1; y_2; y_3)$ respectively. One way to implement those codes is to use constants, that is, presenting ones and zeros at the input of the structure. Also, they could be programmed into the structure using various memory devices [3].

- The input switch matrix is combinational logic that produces all the possible next state entries for each current control input.
- The next state logic consists of an independent path for each of the present states in the state assignment flow table.
- The storage element is a D-FF that preserves the present state.

The operation of the architecture is as follows. The current control input selects the set of potential next states that the circuit can assume (input column in the flow table). The present state variables select the exact next state (row in the flow table) that the circuit will assume at the next clock pulse.

4 Formal Specification

The previous section presented a description of the SISM architecture and operation. This section presents the formal specification of the SISM architecture. The behavioral specification is introduced first and then a structural implementation is described.

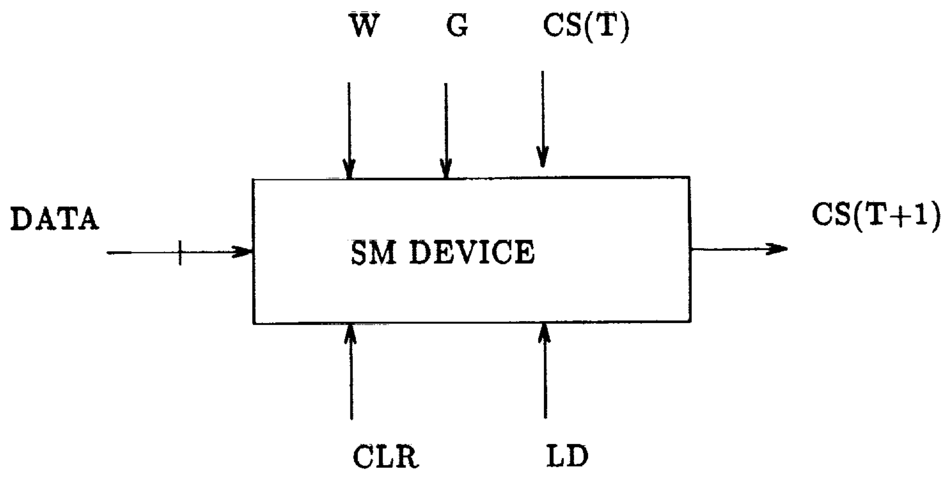

Figure 2: General state machine device

4.1 The Behavioral Specification

A general behavioral description of all state machines can be specified by defining a predicate that relates the inputs and outputs and defines the state transition. Figure 2 shows a general state machine device. The behavior of the state machine device can be specified by a predicate `sism-spec`, that is true only when the combination of the values of the variables w, g, data, clr, ld; and the state variable cs is one that could occur on the corresponding input and output signals of the device. The variables are references to actual signals and data as explained below.

- **'w'**, "(:num)".
  This represents the width of the state machine, i.e., the number of next state variables.

- **'g'**, "(: *time* → *num*)".
  This is the control input to the state machine. It is represented as function associated to time. That is at time (t), the input (g) is the control input which is a number from zero to *I*. Where *I* is the maximum number of control inputs.

- **'data'**, "(: *num* → *num* → *num* → *bool*)".
  This is the destination state codes for the entire state machine. It is represented as a function associated with the width of the state machine and the list of data for each of the next state variables.

- **'clr'**,"(: *time* → *bool*)".
  This signal when enabled will forces the output values to be cleared to low.

- **'ld'**, "(: *time* → *bool*)".
  This signal when enabled will load the input data to the D-ff and present it to the output.

- **'cs'**, "(: *time* → *num* → *bool*)".
  This is the current state value. It is represented as function associated to time. That is at time (t) this value will enable one path from the input to the output.

The overall behavior of the state machine is given by the following logic term:

```
sism-spec =
⊢def  sism_spec w g data clr ld
                        (cs:num->num->bool)  =
      (∀ t:num.    cs (t+1) = (clr t →  ZEROS w |
                   ld t →  data  (g t) (val w (cs t)) |
                                   cs t))"
```

The predicates `sism-spec` asserts that the relationship between those values corresponds to the way the state machine works in practice. That is, the next state of the machine at time (t+1) is a function of the value of the data input and the current state at time (t).

4.2 The Structural Specification

An implementation of state machines based on the sequence invariant architecture is presented. Using tools available in HOL the structure of the SISM can be described by specifying high level descriptions of the major pieces of the SISM device and combining them so that they correspond to the actual structure. The structure of the SISM can be represented by a predicate `sism-imp` with a definition as follows:

```
(sism_imp =
    sism_imp w g data  clr ld cs = (sism_imp_rec w w g data  clr ld cs)"
```

The predicate sism-imp-rec defines the structure of the circuit. The predicate is defined recursively on its width indicating the iterative structure of the circuit. The predicate is defined as follows:

```
 (sism_imp_rec =
   "(sism_imp_rec 0 w g data  clr ld cs = block 0 w g data
                                          clr ld cs)
         ∧
    (sism_imp_rec (n+1) w g data  clr ld cs =
       ((sism_imp_rec n  w g data  clr ld cs)  ∧
        (block (n+1) w g  data  clr ld cs )))"
```

The predicate block gives the structure of a single slice of the circuit. Block is defined by conjoining the predicates that specify the behaviors of each component with the logical connective (∧) and using existential quantification (∃) to hide the internal signals. The following logic term describes block:

```
block =
⊢def   block id w  g data clr ld  cs  =
     (∃ out1 out2.
                      (sel id w g data  out1 )  ∧
                      (mux w out1 cs out2 ) ∧
                      (d_ff out2  ld clr (cs id)))"
```

In this definition the two internal lines (outl; out2) are hidden from the external environment using the existential quantifier (∃). The definition of block states that the values which can appear on the external inputs and outputs of the SISM device are precisely those which satisfy the constraints imposed by the predicates modeling the three modules from which it is built. The modules that are used to define the predicate block are explained next.

The Selector module The selector module is defined using predicates as a function. The predicates that defines the behavior specification is a function as shown below,

```
sel=
⊢def  sel id w g data out =
       ∀ (t:time) line.
       (line < (2 EXP (SUC w))) ⇒
       (out line t) = (data id line (g t))");;
```

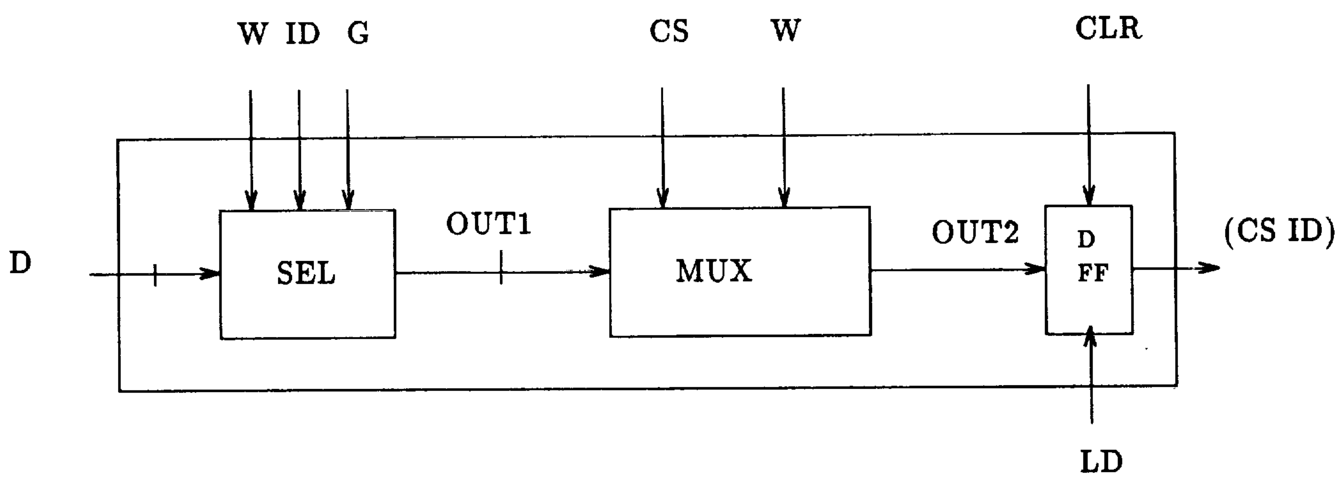

Figure 3: A block representing the SISM device

The selector is a device that is controlled by the control inputs. For each block there are 2^w selectors. Hence, 2^w outputs are presented to the next device. The data input to the selector are the destination state codes. The outputs are all the data selected by the current control input. Referring to the definition and to Figure 3, the selector has three external inputs and one internal output. Some of the variables are described earlier, however the new variables are described as follow.

- 'id', "(:num)".
  This represents the current block of the state machine, i.e. if w=3 and id=1 then the current next state variable is the first variable in the SISM block.

- '**out**', "(: *num* → *time* → *bool*)".
  This function represents all possible outputs for each next state variable under the current control input.

The MUX Module The MUX module is a function that takes 2^w inputs and present one value to the output based on the current state. The following predicate describes the behavior of MUX:

```
mux=
⊢def mux w input cs out =
      (∀ t:time. (out t) = (input (val w ((ABS w cs) t)) t))"
);;
```

Referring to the definition and to Figure 3, the MUX module has two external inputs, one internal input, and one internal output. The internal inputs and outpus are described as follows.

- **'input'**, "(: *num* → *time* → *bool*)".
  This is the data provided by the previous module. It is a bit vector of length 2^w, which represent all possible next state entries.

- **'output'**, "(: *time* → *bool*)".
  This is the value selected by the current state as one of the next state variable at the next clock pulse.

The D-ff Module The D-ff module is a memory device that present the input to the output at the next clock pulse. The predicate that describes the behavior specification is as follows:

```
d-ff=
⊢def d_ff in ld clr q =
       (∀ t:time . q(t+1) = ((clr t) →  F   |
                               (ld t)  →  in t | q t))
     ∧ (q 0 = F)"
);;
```

Referring to the definition and to Figure 3, the following variables are defined,

- **'in'**,"(: *time* → *bool*)".
  This is the next state variable provided by the previous module to be presented to the output at the next clock pulse.

- **'q'**, "(: *time* → *bool*)".
  This is the output value which constitute one of the variables that when combined with the other outputs from the other blocks, result in the current state.

5 Verification

The goal of the verification is stated in logic as follows:

```
    "∀ w g data clr ld cs.
       sism_imp w g data clr ld cs ⇒
       sism_spec w g (DATA_ABS w data) clr ld (ABS w cs)"
```

The goal states that the structural implementation implies the behavioral description of the circuit, or, that the behavior follows from the structure. In the goal, `DATA-ABS` and `ABS` are two functions used to abstract the signals `w`, `data` and `cs` which are defined at the structural level to behavioral level signals.

The verification is approximately 60% done. The proof is carried out using induction on the width of the SISM. HOL provides mechanical support for induction, rewriting, case analysis and other necessary proof techniques.

6 Conclusion

This paper presents the design for a SISM that is being proven to work correctly. This is especially significant because the design of the SISM is very general. Future work will entail tying the structural specification to the actual circuit and using this work to verify specific state machines based on the SISM design.

References

[1] Paul Loewenstein, "Reasoning about State Machines in Higher-Order Logic", In M. Leeser and G. Brown, editors, Workshop on Hardware Specification, Verification, and Synthesis, Mathematical Aspects, SpringerVerlag 1989.

[2] S. Whitaker, G. Maki, and M. Shamanna, "Reliable VLSI Sequential Controllers", NASA Space Engineering research Center, Symposium on VLSI 1990. University of Idaho.

[3] G. Birtwistle and P. A. Subrahmanyam, Editors. "Current Trends In Hardware Verification And Automated Theorem Proving", Springer-Verlag New York Inc. 1989. pp 4-19.

[4] M. Alahmad, "Reconfigurable Sequence Invariant State Machine", Masters Thesis. University of Idaho. Dec. 1991.