Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation[*]

Randal E. Bryant
Derek L. Beatty
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA

Carl-Johan H. Seger
Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1W5
Canada

Abstract

Symbolic trajectory evaluation is a new approach to formal hardware verification combining the circuit modeling capabilities of symbolic logic simulation with some of the analytic methods found in temporal logic model checkers. We have created such an evaluator by extending the symbolic switch-level simulator COSMOS. This program gains added efficiency by exploiting the ability of COSMOS to evaluate circuit operation over a ternary logic model, where the third value *X* represents an unknown logic value. This program can formally verify systems containing complex features such as switch-level models, detailed timing, and pipelining.

1. Introduction

Formal verification seeks to overcome the weakness of informal design testing by simulation. Using our verifier, one can prove that a switch-level model of the transistor circuit correctly implements a formal description of the desired behavior for all possible system operations. Formal verification becomes increasingly desirable as system designs become more complex. With the introduction of pipelining and concurrently-operating subsystems, it becomes increasingly difficult using informal methods to evaluate the many subtle interactions between logically unrelated system activities.

In this paper we describe a new approach to formal verification that augments the circuit modeling capabilities of symbolic, switch-level simulation with some of the analytic capabilities found in temporal logic model checkers. With this increased analytic capability, we can express such temporal aspects of circuit behavior as clocking methodology and the skewing in time caused by pipelining. This paper describes the operation and application of our evaluator using a number of circuit design examples. A detailed presentation of the formal logic is presented in [2].

2. Heritage

Most automated approaches to formal verification (as opposed to more manual methods based on theorem proving) are based either on *symbolic simulation* or on *state machine analysis*. A symbolic simulator evaluates circuit behavior using symbolic variables (typically Boolean-valued) to encode a range of circuit operating conditions. In one simulation run, a symbolic simulator can compute what would require many runs of a conventional simulator. Symbolic simulation can support detailed circuit models and can handle large circuits. In addition to a method for evaluating circuit operation, however, formal verification requires a methodology for specifying the desired behavior and for checking the correspondence between the desired and realized behaviors. Straightforward symbolic simulation is adequate for verifying combinational circuits or the combinational portion of sequential circuits [5]. For verifying sequential systems where the state storage elements are not identified, or for which the behavioral specification is not based on the explicit state encoding, however, a more powerful methodology is required.

With state machine analysis, the program creates a finite state machine representation of the circuit. The program can then analyze properties of the machine, such as deciding the truth of a temporal logic formula [1] or determining whether two finite state machines are equivalent [4]. These methods are very powerful in their analytic capability. Their major limitation is in their performance as the finite state machines become very large. Recent versions of these programs encode the states symbolically and hence can analyze systems having very large numbers of states. For circuits involving large amounts of storage, such as memories, data paths, and processors, the automata become too large to represent even symbolically.

3. Symbolic Trajectory Evaluation

Symbolic trajectory evaluation extends symbolic simulation with some of the analytic capability of finite state system analyzers. The user specifies the desired behavior of the system by assertions expressed as temporal logic formulas. Our temporal logic is quite restricted; it allows us to express properties of the circuit over *trajectories*: bounded-length sequences of circuit states. Our program verifies these formulas by a modified form of symbolic simulation, avoiding the need to extract a finite state machine representation. Furthermore, we exploit the 3-valued modeling

[*] This research was supported by the Defense Advanced Research Project Agency, ARPA Order Number 4976, and by the National Science Foundation under grant number MIP-8913667.

capability of the simulator, where the third logic value *X* indicates an unknown or indeterminate value. By judicious use of *X* as "don't care" values, we can reduce the complexity of the symbolic manipulations considerably. For example, as will be shown, we can verify the correctness of a data path containing m registers of n bits each by performing a symbolic evaluation involving just $O(n + \log m)$ variables, whereas approaches based on symbolic finite state machine analysis require manipulations of functions involving $O(nm)$ Boolean variables. Finally, since our verifier is based on simulation, we can more easily model timing details that are normally abstracted away by state machine models.

Our verifier supports a methodology in which the user expresses the desired system behavior as a set of assertions about the state transitions of an abstract state machine. In addition, the user provides temporal formulas defining such circuit details as the clocking methodology, the timing of input and output signals, and how the circuit realizes the abstract state both spatially and temporally. This form of specification works well for circuits that are normally viewed as *state transformation systems*, i.e., where each operation is viewed as updating the circuit state. Examples of such systems include memories, data paths and processors. For such systems, the complex analysis permitted by state machine analyzers is not required.

4. Specifying Circuit Behavior

4.1. Specification Logic

We model a circuit as operating over logic levels 0, 1, and a third level *X* representing an indeterminate or unknown level. These values can be partially ordered by their "information content" as $X \sqsubseteq 0$ and $X \sqsubseteq 1$, i.e., *X* conveys no information about the node value, while 0 and 1 are fully defined values. The only constraint we place on the circuit model—apart from the obvious requirement that it accurately model the physical system—is monotonicity over the information ordering. Intuitively, changing an input from *X* to a binary value (i.e., 0 or 1) must not cause an observed node to change from a binary value to *X* or to the opposite binary value. In extending to symbolic evaluation, the circuit nodes can take on arbitrary ternary functions over a set of Boolean variables $\mathcal{V}$.

Symbolic circuit evaluation can be thought of as computing circuit behavior for many different operating conditions simultaneously, with each possible assignment of 0 or 1 to the variables in $\mathcal{V}$ indicating a different condition. Formally, this is expressed by defining an *assignment* ϕ to be a particular mapping from the elements of $\mathcal{V}$ to binary values. A formula F in our logic expresses some property of the circuit in terms of the symbolic variables. It may hold for only a subset $\mathcal{D}$ of the possible assignments. Such a subset can be represented as the Boolean domain function d over $\mathcal{V}$ yielding 1 for precisely the assignments in $\mathcal{D}$. The constant functions **0** and **1**, for example, represent the empty and universal sets, respectively.

Our verifier analyzes the temporal behavior of the circuit at a *phase* level, where one phase represents a period of time in which all external inputs are held fixed and the circuit operates until it reaches a stable state. This timing level was chosen to allow more detailed analysis than traditional state machine models, which represent an entire clock cycle as a single state transition. Potentially, our method could be extended to even more detailed timing models, including ones modeling real time.

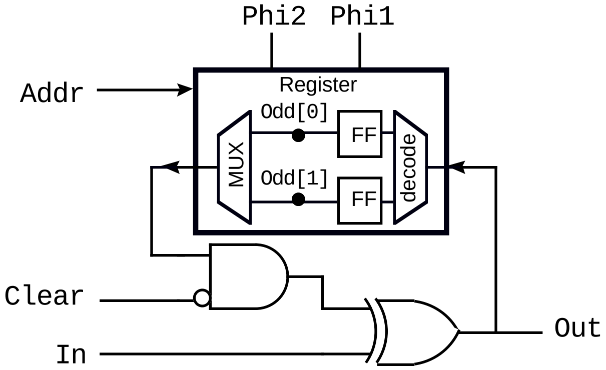

Figure 1: **Addressable Serial Parity Circuit.** Parity is maintained for two multiplexed signals.

Our algorithm checks only one basic form, the *assertion*, in the form of an implication $[A \implies C]$; the antecedent A gives the stimulus and current state, and the consequent C gives the desired response and state transition. System states and stimuli are given as trajectories over fixed length sequences of phases.

We describe each of these trajectories with a temporal *formula*. Primitive formulas specify Boolean values for circuit nodes, i.e., $\mathtt{n} = 1$ or $\mathtt{n} = 0$, and express the property that node `n` has the specified value for the entire phase. The only combining form is conjunction $F_1 \wedge F_2$, and the only temporal operator is the "next-phase" operator $\mathbf{P}F$, stating that F must hold in the following phase. This operator is similar to the next-time operator $\mathbf{X}$ found in linear temporal logic [6]. In addition, a restriction operator creates a formula $B \rightarrow F$ stating that the property represented by formula F need only hold for those assignments satisfying Boolean expression B.

The temporal logic supported by our evaluator is far weaker than that of other model checkers. It lacks such basic forms as disjunction and negation, along with temporal operators expressing properties of unbounded state sequences. The logic was designed as a compromise between expressive power and ease of evaluation. It is powerful enough to express the timing and state transition behavior of circuits, while allowing assertions to be verified by an extended form of symbolic simulation.

4.2. Specification Example

We illustrate our methodology with the addressable serial parity circuit shown in Figure 1, designed to maintain the parity of two channels multiplexed onto a single line. On each cycle, the channel to be updated is specified by the `Addr` input. Setting input `Clear` to 1 has the effect of setting the previous parity value for the channel to 0. Internally, the circuit consists of some combinational logic, plus a two-bit register file. Circuit nodes `Odd[0]` and `Odd[1]` are identified as representing the parity values for the two channels.

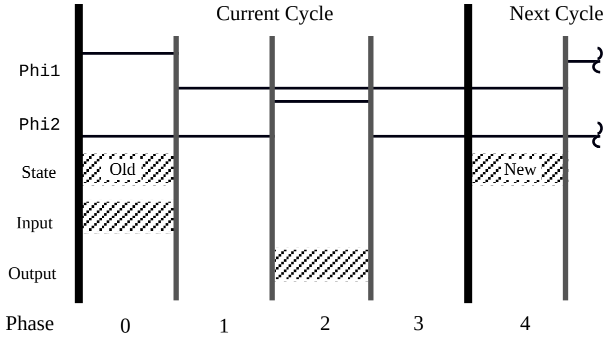

Figure 2: **Circuit Timing**. Each clock cycle is modeled as four phases, with the state, input, and output data valid in the phases indicated. Verification requires evaluating a trajectory of 5 phases.

The desired behavior of this circuit can be expressed by a single assertion stating that each register should be updated appropriately when it is addressed and should hold its value when it is not. In developing this assertion, we incorporate the timing information illustrated in Figure 2. Each cycle consists of 4 phases including the nonoverlapping periods of the clock nodes. Inputs are applied during the initial phase, and the output is guaranteed to be valid two phases later. Furthermore, the internal state is held stable during the initial phase. Thus, we must evaluate circuit operation over a trajectory of 5 phases—one full clock cycle plus the first phase of the next. These phases are numbered from 0 to 4 to match the nesting of next-phase operators in the specification.

Informally, the assertion states:

- Given:
  - The clocks are cycled correctly.
  - Inputs values a, c, and u are applied during phase 0 to `In`, `Clear`, and `Addr`, respectively.
  - Node `Odd[`v`]` has value b during phase 0
- Then, for the case where $u = v$:
  - Value $a \oplus b\overline{c}$ will appear on `Out` during phase 2
  - Node `Odd[`v`]` will have this same value in the initial phase of the following cycle (phase 4).
- Otherwise ($u \neq v$):
  - Node `Odd[`v`]` will have value b in the initial phase of the following cycle.

Through the use of symbolic variables a, b, c, u, and v, we have expressed the effect of many different operations with a single assertion. Note that we do not attempt to specify the circuit behavior when the clocks are not cycled properly, when inputs are not applied at the correct times, etc.

We construct the antecedent by first defining the operation of the clocks. As shorthand, define formulas representing the possible signals applied to the clocks:

$$\begin{aligned} \mathit{Clk10} &\overset{\text{def}}{=} \texttt{Phi1} = 1 \wedge \texttt{Phi2} = 0 \\ \mathit{Clk00} &\overset{\text{def}}{=} \texttt{Phi1} = 0 \wedge \texttt{Phi2} = 0 \\ \mathit{Clk01} &\overset{\text{def}}{=} \texttt{Phi1} = 0 \wedge \texttt{Phi2} = 1 \end{aligned}$$

Define *Clocks* as describing the clocking behavior over the entire trajectory:

$$\mathit{Clocks} \overset{\text{def}}{=} \mathit{Clk10} \wedge \mathbf{P}\mathit{Clk00} \wedge \mathbf{P}^2\mathit{Clk01} \wedge \mathbf{P}^3\mathit{Clk00} \wedge \mathbf{P}^4\mathit{Clk10}$$

where $\mathbf{P}^i$ denotes i repetitions of the next-phase operator.

We can then write the specification as:

$$\begin{aligned} &\mathit{Clocks} \wedge (\texttt{In} = a) \wedge (\texttt{Clear} = c) \wedge (\texttt{Addr} = u) \wedge \\ &(\texttt{Odd[}v\texttt{]} = b) \\ &\quad\Longrightarrow \\ &u = v \rightarrow \left(\mathbf{P}^2(\texttt{Out} = a \oplus b\overline{c}) \wedge \mathbf{P}^4(\texttt{Odd[}v\texttt{]} = a \oplus b\overline{c})\right) \wedge \\ &u \neq v \rightarrow \left(\mathbf{P}^4(\texttt{Odd[}v\texttt{]} = b)\right) \end{aligned}$$

We use several abbreviations to keep the specification concise. The notation $\texttt{n} = a$ stands for the formula: $(a \rightarrow \texttt{n} = 1) \wedge (\overline{a} \rightarrow \texttt{n} = 0)$. The notation $\texttt{Odd[}v\texttt{]} = b$ stands for: $(\overline{v} \rightarrow \texttt{Odd[0]} = b) \wedge (v \rightarrow \texttt{Odd[1]} = b)$.

5. Verifying Circuits

5.1. Evaluation Algorithm

The constraints we place on assertions make it possible to verify an assertion by a single evaluation of the circuit over a number of phases determined by the deepest nesting of the next-phase operators. In essence, we simulate the circuit over the unique weakest (in information content) trajectory allowed by the antecedent, while checking that the resulting behavior satisfies the consequent. In this process we compute a Boolean function *OK* expressing those assignments for which the assertion holds. For a correct circuit, this function should equal **1**; otherwise, we can determine which cases failed by examining it.

More precisely, we first rewrite the antecedent into a form $A_0 \wedge \mathbf{P}A_1 \wedge \mathbf{P}^2A_2 \wedge \cdots \wedge \mathbf{P}^kA_k$, where each component A_i is *instantaneous*, i.e, it does not contain any next-phase operators. We rewrite the consequent similarly. Due to our restricted formula syntax, each such instantaneous formula obeys the property that for any assignment ϕ, one of the following cases must hold:

1. The formula has an internal inconsistency (e.g., $\texttt{n} = a \wedge \texttt{n} = b$ for assignments where $a \neq b$).
2. There exists a unique circuit state, minimal in information content, satisfying the formula.

We can combine this information for all possible assignments symbolically by associating with each instantaneous formula F a Boolean function OK_F denoting those assignments where the formula has no internal inconsistencies, and a symbolic state vector $\vec{a}_F$ which for a given assignment describes the minimal circuit state if the formula is consistent, and has all elements equal to *X* otherwise.

As an example, consider the instantaneous formula F defined as $\texttt{Odd[}u\texttt{]} = a \wedge \texttt{Odd[}v\texttt{]} = b$. The associated domain function is $\mathit{OK}_F = \overline{(a \oplus b)} + (u \oplus v)$, i.e., the formula is consistent so long as we do not try to assign opposite binary values to the same node. The minimal state values assigned to nodes `Odd[0]` and `Odd[1]` would be ternary functions over the variables a, b, u, and v given by the tables:

Odd[0]	*u v*			
a b	00	01	10	11
00	0	0	0	Ⓧ
01	$\boxed{X}$	0	1	$\boxed{X}$
10	$\boxed{X}$	1	0	$\boxed{X}$
11	1	1	1	Ⓧ

Odd[1]	*u v*			
a b	00	01	10	11
00	Ⓧ	0	0	0
01	$\boxed{X}$	1	0	$\boxed{X}$
10	$\boxed{X}$	0	1	$\boxed{X}$
11	Ⓧ	1	1	1

□ Overconstrained

○ Unspecified

Observe that some of the *X* entries in these tables arise due to overconstrained cases, where OK_F yields 0. Others arise due to unspecified cases. For example, when $u = v = 1$, no value is specified for node `Odd[0]`.

Starting from a setting of the inputs and an initial state of the internal nodes, a simulator such as COSMOS simulates one phase of circuit operation by repeatedly updating the internal nodes according to their state excitation values until a stable state is reached. This process can be viewed as the cycling of a finite state automaton with the new values of the internal nodes computed according to their excitation functions, and the new values of the input nodes equal to their old values.

Symbolic trajectory evaluation is implemented by modifying this algorithm. For each phase i of the trajectory, the antecedent component A_i defines some constraints on the input and internal node states. These constraints are imposed by setting each element of the circuit state to the least upper bound (denoted by $\sqcup$) of the current value and the corresponding element of $\vec{a}_{A_i}$, where this operation has the function table:

$\sqcup$	0	1	*X*
0	0	–	0
1	–	1	1
X	0	1	*X*

Entries labeled '–' indicate cases where the operation is overconstrained, i.e., we are attempting to assign opposite binary values to a single node. As the evaluation progresses, the program keeps track of the assignments that do not have overconstrained behavior, represented by a Boolean function *Traj*.

Unlike a simulator, the evaluator will set an input node to *X* whenever it is not constrained by the antecedent. In this way, the evaluator makes no assumptions about input values except those explicit in the antecedent.

The consequent component C_i defines some checks that should be made of the circuit state during phase i. We require that each component of the observed circuit state remain greater or equal to the corresponding element of $\vec{a}_{C_i}$ throughout the phase. Thus, the program keeps track of the assignments that satisfy these checks, represented by a Boolean function *Check*. The program also computes function OK_A (respectively, OK_C), representing the assignments where the antecedent (resp., consequent) contains no inconsistencies. Once the circuit has been evaluated for its entire trajectory, the program computes the function $OK = \overline{OK_A} + \overline{Traj} + (OK_C \cdot Check)$, representing the set of assignments for which the entire assertion is valid, i.e., where the consequent holds whenever the antecedent can be established.

One key property of our evaluation algorithm is that it involves symbolic manipulation only over those variables explicit in the assertion. In contrast, symbolic state machine analyzers must perform manipulations involving as many variables as there are bits of state in the circuit. This difference can be quite significant.

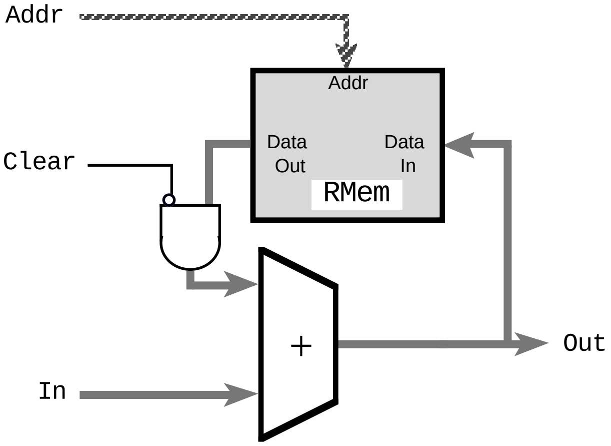

Figure 3: **Addressable Accumulator.** Sums can be accumulated in m different registers.

5.2. Implementation

We constructed our verifier by extending the COSMOS symbolic switch-level simulator [3]. The simulator supports a three-valued circuit model by encoding the values 0, 1, and *X* as pairs of binary values. Ternary values are represented by pairs of Ordered Binary Decision Diagrams (OBDDs), according to this encoding; ternary operations are implemented similarly.

The assertion syntax we have introduced is rather primitive. To facilitate generating more abstract notations, we have developed a front end to the evaluator by embedding a specification language in Scheme, a dialect of Lisp. This language allows the user to write formulas in terms of vectors of Boolean values, apply operations such as binary arithmetic and bitwise logical operations, and extend the language. These formulas are then expanded into Boolean operations automatically, generating a command file for the evaluator.

6. Verifying Complex Circuits

We now illustrate verification of more complex sequential circuits, including pipelining. Consider the addressable accumulator shown in Figure 3. This circuit can maintain the sum of signals for m different channels, storing the sums in its register array. Timing is similar to that of the parity circuit.

A specification of this circuit has the same general form as the specification for the serial parity circuit. Instead of using single Boolean variables (e.g., u, a) to indicate possible address and data values, however, we use vectors of Boolean variables (e.g., $\vec{u}$, $\vec{a}$). The desired output and new state values are defined in terms of the binary representation of the addition function. Finally, rather than fixing the nodes representing the stored values in the abstract specification, we write the assertions in terms of an abstract predicate Reg, where Reg$[\vec{u}, \vec{a}]$ states that value $\vec{a}$ is held in some abstract register $\vec{u}$. The mapping between the abstract state and the actual circuit state is specified by defining this predicate in terms of values on the circuit nodes.

For the circuit illustrated in Figure 3, the circuit register ar-

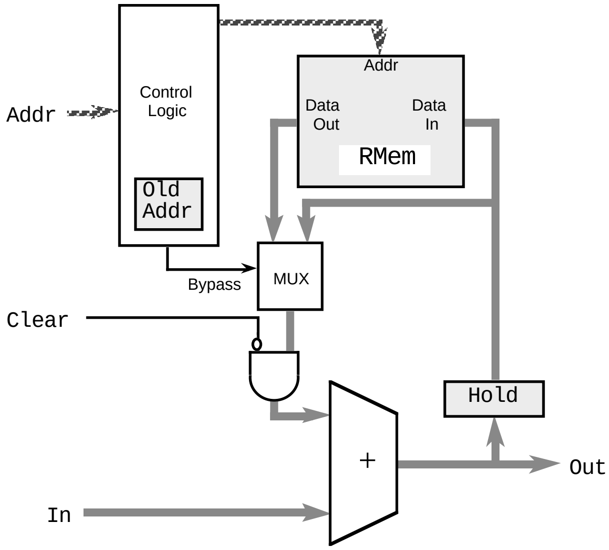

Figure 4: **Pipelined Addressable Accumulator Circuit.** Register access and adder operation occur simultaneously. Bypass logic handles the case where the same address is used on consecutive operations.

ray RMem is in exact correspondence with the abstract system registers. Thus, we can define the state mapping as $\mathrm{Reg}[\vec{u}, \vec{a}] \quad \stackrel{\mathrm{def}}{=} \quad (\texttt{RMem}[\vec{u}] = \vec{a})$, using a vector extension of the indexing notation introduced earlier.

6.1. Pipelined Circuits

Pipelining enhances circuit performance by increasing the amount of concurrent activity. Most pipelined systems are designed to be transparent to the outside. That is, interlock and bypass circuitry makes the system appear to be unpipelined. The difficulty of designing such systems makes them an important class for formal verification. In verifying such a system, we want to prove that the pipelined system realizes an unpipelined specification, and hence the behavior specification in terms of abstract system state should not reflect the pipeline structure. Instead, we include this information in the state mapping.

Consider, for example, a pipelined version of the addressable accumulator illustrated in Figure 4. This circuit exhibits the same external behavior as the one illustrated in Figure 3, but achieves greater performance by overlapping the adder and register write operations. That is, on each cycle a value is first read from the register file. Then, while the adder is computing the sum for the current cycle, the value from the previous cycle is written into the register array. When the same address is used in succession, the previous adder output, stored in register Hold, is transferred directly to the adder input. This bypassing is implemented by control logic that includes a register OldAddr storing the address from the previous cycle.

The abstract specification for this circuit is identical to that of the unpipelined circuit. The state mapping, however, is quite different. The contents of abstract register $\vec{u}$ may be in either the register file or in Hold, depending on the previous address. This is expressed by defining the register predicate $\mathrm{Reg}[\vec{u}, \vec{a}]$ as:

Circuit	Num. of Regs.	Word Size	Num. of Trans.	CPU Secs.	Mega-bytes
Adder	–	32	1186	7	0.2
SRAM	32	32	6666	92	1.0
Accum.	2	32	2352	22	0.3
	32	16	4603	290	1.0
	16	32	5665	267	1.8
	32	32	8332	883	1.9
Pipelined	2	32	2823	21	0.4
Accum.	32	32	8875	591	1.9

Table 1: **Verifier performance for example circuits.** Measured on SUN-4/110.

$$\exists \vec{w} \left[\begin{array}{lll} \texttt{OldAddr} = \vec{w} & \wedge & (\vec{u} = \vec{w} \;\rightarrow\; \texttt{Hold} = \vec{a}) \\ & \wedge & (\vec{u} \neq \vec{w} \;\rightarrow\; \texttt{RMem}[\vec{u}] = \vec{a}) \end{array} \right]$$

In this mapping, we use a vector of existentially quantified variables $\vec{w}$ to indicate that register OldAddr may hold an arbitrary address independent of the current operation. Given such an address, abstract register $\vec{u}$ maps to either the holding register ($\vec{u} = \vec{w}$), or to register $\vec{u}$ in the register file ($\vec{u} \neq \vec{w}$). Adding quantified variables to our assertion logic is straightforward. The quantifiers map directly into operations on the OBDDs.

7. Experimental Results and Observations

Table 1 shows the performance of our verifier running on a SUN-4/110 for some of the circuits discussed above. As the table indicates, we separately verified the adder and register file (implemented with a static RAM) before assembling the accumulators. In our experience, verification is much more manageable when conducted as the designs are constructed. Attempting to verify existing designs created by other people without precise information about interface timing and state encoding tends to be time-consuming and frustrating. Often, a fair amount of reverse engineering is required.

As the figures indicate, these circuits are well within the capabilities of our program. As the circuits grow larger by increasing either the word size or the number of registers, the time required scales less than quadratically, while the memory required scales at most linearly. It is interesting to note that the pipelined accumulator can be verified more quickly than the unpipelined accumulator. This is due to particular features of the assertion and state mapping. More typically, complex pipelines require somewhat more CPU time to verify than do simpler circuits.

To gain an understanding of how this form of verification scales to larger designs, we verified switch level implementations of a family of pipelined data paths containing a register file, ALU, pipe registers, and a simple controller.[1] The data path processes each instruction in four pipe stages: instruction read, operand read, execute, and write back. Such a pipeline has a *read-after-write*

[1] K. McMillan designed the data path, and R. Luthi implemented it at the switch level.

Word	Registers				
Size	2	4	8	16	32
2	60s	109s	120s	178s	320s
	0.5M	0.5M	0.7M	0.8M	1.1M
4	196s	274s	288s	449s	669s
	1.0M	1.1M	1.1M	1.8M	1.9M
8	323s	398s	514s	704s	1116s
	1.7M	1.8M	1.8M	3.2M	3.3M
16	1119s	1289s	1827s	2290s	
	5.9M	6.0M	6.0M	6.1M	

Table 2: **Verifier Performance for Pipelined Data Paths**. Measured on DECstation 3100.

data hazard. When an instruction i reads a register which is the destination register of instruction $i-1$ or $i-2$, those instructions $i-1$ and $i-2$ have not yet written back their results into the register file. The controller detects this hazard and compensates using two features. First, the register file write operations occur in the first half of a clock cycle, while read operations occur in the second half, so the controller can write a value into a register and read the newly written value from the same register in one clock cycle. Second, the data path contains a register bypass to pass the result of one instruction directly to the following instruction via the appropriate pipe register. The controller also includes logic to detect a no-operation op code and inhibit the appropriate writeback pipe stage. The ALU implements 10 operations.

Our specification of the data path consists of two kinds of assertions. First, we separately specified each ALU operation, and second, we specified that the NO-OP instruction should preserve register state. Space precludes a more detailed description of this specification.

Performance of our prototype verifier on the data path example is shown in Table 2. Note that execution time scales less than quadratically with word size and less than linearly with the number of registers. The memory requirement scales even more gradually. Attempting to verify larger data paths overloaded the Scheme interpreter we were using to implement the front end interface.

8. Conclusions

Our experience to date indicates that this is a promising approach for verifying circuits viewed as state transformation systems. It can operate on detailed switch-level models and verify timing at the phase level. It can be used to verify pipelined systems using high-level specifications that express the desired behavior without explicitly referencing the pipeline structure. It requires the definition of both assertions and implementation mappings.

For future work, we plan to extend our program, improve its performance, and test it on more ambitious circuit designs such as microprocessors. Although we have found an embedded language to provide powerful abstraction mechanisms for constructing behavioral specifications and state mappings hierarchically, our current configuration of running two separate programs is not ideal. The command files produced by the front end can be extremely large, and it is difficult to relate errors from the verifier back to the part of the original specification that failed. Furthermore, some form of graphical interface to specify the input, output, and clock timing would be helpful.

References

[1] J. R. Burch, E. M. Clarke, D. L. Dill, and K. McMillan, “Sequential Circuit Verification using Symbolic Model Checking,” *27th Design Automation Conference*, June, 1990.

[2] R. E. Bryant, and C.-J. H. Seger, “Formal Verification of Digital Circuits Using Symbolic Ternary System Models,” *Workshop on Computer-Aided Verification*, Rutgers, NJ, June, 1990.

[3] K. Cho, and R. E. Bryant, “Test Pattern Generation for Sequential MOS Circuits by Symbolic Fault Simulation,” *26th ACM/IEEE Design Automation Conference*, June, 1989, pp. 418–423.

[4] O. Coudert, J.-C. Madre, and C. Berthet, “Verifying Temporal Properties of Sequential Machines without Building their State Diagrams,” *Workshop on Computer-Aided Verification*, Rutgers, NJ, June, 1990.

[5] J. C. Madre, and J. P. Billon, “Proving Circuit Correctness using Formal Comparison between Expected and Extracted Behaviour,” *25th ACM/IEEE Design Automation Conference*, 1988.

[6] A. Pnueli, “The Temporal Logic of Programs,” *18th Symposium on the Foundations of Computer Science*, IEEE, 1977, pp. 46–56.