Sequential Circuit Verification Using Symbolic Model Checking

J. R. Burch E. M. Clarke K. L. McMillan
Carnegie Mellon University

David L. Dill
Stanford University

Abstract

The temporal logic model checking algorithm developed by Clarke, Emerson, and Sistla [9] is modified to represent a state graph using *binary decision diagrams* (BDD's) [4]. Because this representation captures some of the regularity in the state space of sequential circuits with data path logic, we are able to verify circuits with an extremely large number of states. We demonstrate this new technique on a synchronous pipelined design with approximately 5×10^{20} states. Our model checking algorithm handles full CTL with fairness constraints. Consequently, we are able to handle a number of important liveness and fairness properties, which would otherwise not be expressible in CTL. We give empirical results on the performance of the algorithm applied to both synchronous and asynchronous circuits with data path logic.

1 Introduction

Bugs found late in the design phase of a digital circuit are a major cause of unexpected delays in the realization of the circuit in hardware. This has stimulated interest in formal verification techniques for hardware designs. A number of different techniques have been proposed, but nearly all can be classified in terms of the natural division between the *data paths* and the *controlling circuitry* in digital devices. The most successful methods to date for verifying data path logic treat only functional behavior, without considering sequential behavior. These methods are frequently based on the use of automatic theorem provers or proof checkers and may require considerable assistance from the user in constructing a correctness proof. The most effective techniques for reasoning about sequential behavior (we use the term sequential behavior to include the behaviors of concurrent systems, as well) on the other hand, usually require a complete exploration of the state space of the circuit. The state enumeration techniques are attractive, because they are highly automatic: the user simply provides a description of the circuit implementation and its specification; the system does the rest. In the case of a single controller, the approach is often quite practical, since the number of states tends not to be excessively large. The approach has not been very useful with data path circuits, since the number of states is almost always too large to permit explicit enumeration. In order to reason about the complex interaction between controllers and data paths, however, we need techniques that are able to handle both types of circuits.

In this paper, we show how a technique for reasoning about sequential circuits, called *temporal logic model checking* [8, 9], can modified to represent a state graph using *binary decision diagrams* (BDD's) [4]. Because this representation captures some of the regularity in the state space determined by the data path logic, we are able to verify sequential circuits with an extremely large number of states. The algorithm is based on computing fixed points of functions from sets of states to sets of states (predicate transformers). We can express both the sets of states and the transition relations in terms of BDD's. Thus, we are able to avoid explicitly constructing the state graph of the circuit. We have tested the performance of the algorithm on both synchronous and asynchronous (self-timed) circuits with data path logic.

Previously, most of the applications of BDD's have been to the verification of combinational circuits. However, there have been some recent applications to sequential circuits. Bryant [5] uses a symbolic switch-level simulator, in which a sequence of operations is simulated with symbolic inputs. The use of symbolic inputs allows one to verify that certain pre- and post-conditions are satisfied independently of the actual input values applied. A second approach due to Bose and Fisher [2] verifies a pipeline circuit with respect to a simpler abstract model by means of a representation function, in analogy to abstract data type verification.

While both of these approaches are quite powerful for

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976. The National Science Foundation also sponsored this research effort under contract numbers CCR-8722633 and MIP-8858807. The third author is supported by an AT&T Bell Laboratories Ph.D. Scholarship. The fourth author is supported by a CIS Seed Research Grant.

reasoning about certain classes of circuits, they clearly require much more effort from the user than state enumeration methods. In each of these approaches, the user must give a step-by-step specification using pre-condition, post-condition notation, instead of describing the behavior over time with a single temporal formula. The method of Bose and Fisher also requires that the user provide the analog of a data type invariant. An even more serious drawback stems from the limited expressive power of ordinary propositional logic for this type of application. Since they are unable to express unbounded execution histories in propositional logic, their techniques cannot be easily extended to systems of controllers that operate concurrently, nor can they deal with *liveness* properties, which state that an event must occur at some point in the future but do not provide an explicit time bound on when the event should occur.

Coudert, Berthet, and Madre describe a BDD-based system for showing equivalence between deterministic finite automata [10]. Their system performs a *symbolic breadth-first execution* the state space determined by of the product of the two automata. The set of reachable states is represented using a BDD, and in this sense, their method is closely related to our own. However, unlike the technique described in this paper, their method does not deal with indeterminate computations, asynchronous circuits or liveness properties.

Fujita and Fujisawa [12] describe a verification procedure based on linear temporal logic that uses binary decision diagrams to represent the transition conditions in automata derived from temporal logic formulas. However, their technique still suffers from a form of the state explosion problem, because they represent states explicitly in automata derived from temporal formulas. In our work, as in the work by Coudert, Berthet and Madre, boolean decision diagrams are used to represent both the transition relation of the model and subsets of the state space, so that the state graph is never explicitly constructed.

Recently, Fisher and Bose [1] have described a BDD based algorithm for CTL model checking that is applicable to synchronous circuits. However, their algorithm does not support fairness constraints [9], so it is of limited use in proving liveness properties. Also, they do not provide empirical results on the algorithm's performance.

2 CTL and Model Checking

The logic that we use to specify circuits is a propositional temporal logic of branching time, called CTL or Computation Tree Logic [9]. The formulas of the logic describe properties of *computation paths.* For our purposes, a computation path is the infinite sequence of states encountered by a circuit during some sequential execution. In addition to the usual logical connectives $\neg$ and $\wedge$, the logic has four operators for expressing temporal relationships. The *next time operator* **X** indicates a condition that holds in the next state of a computation. Thus, if f if a formula in CTL, then the formula $\mathbf{X}f$ holds of a computation path p if f holds in the immediate successor of the first state in p. The **G** operator denotes a property that holds globally in all states of a computation path. The **F** operator denotes a property that holds sometime in the future. The *until* operator $f\mathbf{U}g$ holds of a computation path p if there exists a state s on p where g holds, and if f holds in all the states preceding s.

In general, more than one possible computation path may begin at a given state. When a temporal operator is prefixed by the *universal path quantifier* **A**, it indicates that the temporal property must hold over all possible computation paths beginning in the current state. Thus, $\mathbf{AX}f$ holds in a state if f holds in all possible next states, while $\mathbf{AG}f$ holds in a state if f holds globally along all possible computation paths beginning with that state. The *existential path quantifier* **E** indicates that the condition expressed by the operator it prefixes holds along *some* computation path beginning with the current state. Formulas involving the universal path quantifier can be expressed using the existential path quantifier and *vice versa.* For example, $\mathbf{AX}f$ is equivalent to $\neg\mathbf{EX}\neg f$.

For the purpose of analyzing digital circuits, it is convenient to assume that the state of a computation is given by a vector of binary state variables $\bar{v}_s$. Each of these variables corresponds to an atomic proposition in the logic. So for example, if v is a state variable in $\bar{v}_s$, then $\mathbf{EX}v$ is true true of a state s if and only if the value of v is 1 in some immediate successor of s. Given a binary transition relation on states, there is an efficient algorithm for determining whether a given formula holds in a given state which is linear in the number of states, and in the length of the formula [9]. However, in the worst case the number of states that can be reached may be 2^n, where n is the number of state variables, hence the procedure may be impractical for circuits with a large number of state variables.

3 Binary Decision Diagrams

This section gives a short description of Bryant's [4] *Binary Decision Diagrams* (BDD's). The rooted, directed acyclic graph in Figure 3 is an example of a BDD representing a boolean function $f(a, b, c, d)$. The following rule can used to see that $f(1, 0, 1, 1) = 1$: trace a path from the root of the diagram to a leaf, at every node choose the branch dictated by the value of the corresponding variable. This rule can be used to completely determine the function represented by a BDD.

A given function can be represented by many differ-

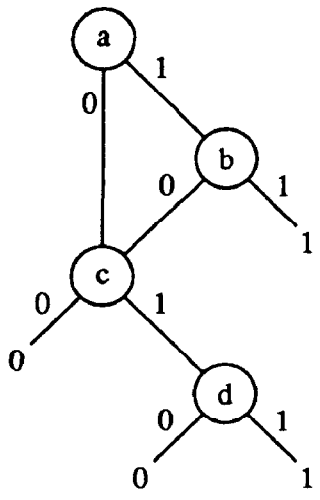

Figure 1: A Binary Decision Diagram

ent DAG's. Bryant placed restrictions on the form of BDD's so that any function has exactly one canonical BDD. One of these restrictions is that a total ordering is given for the variables in the Boolean function. The variable ordering in Figure 3 is $a < b < c < d$. The ordering of the variables down any path from the root of the BDD to a leaf must be consistent with this ordering. Altering the variable order can have a major impact on the size of the BDD needed to represent a given function. BDD's can be implemented in such a way that checking if two BDD's represent the same function can be done in constant time.

Bryant described algorithms for doing basic operations on BDD's such as boolean connectives ($\wedge$, $\vee$, etc.) and functional composition. An algorithm for computing restrictions of functions is also given. The restriction of the function $f(a,b,c,d)$ to $a = 0$ (written $f|_{a=0}$) is the 3-ary function $g(b,c,d) = f(0,b,c,d)$. It is also possible to quantify over boolean variables. For example, the formula $\exists a\, f(a,b,c,d)$ is equal to

$$f(a,b,c,d)|_{a=0} \vee f(a,b,c,d)|_{a=1}.$$

4 Symbolic Model Checking

Model checking means determining whether a given formula f is satisfied in a state given a transition relation R. In this section, we present a model checking algorithm for CTL which uses BDD's as its internal representation, in order to avoid enumerating the elements of the model. The algorithm is defined by a function BDD which recurses over the structure of the formula. The function BDD takes two arguments: a formula f and a representation R of the transition relation. It returns a BDD with the following property: $\textsc{Bdd}(f,R)$ is true in a given state if and only if the formula f is true in that state.

The representation of the transition relation is a BDD $R(\bar{v}_i, \bar{v}_f)$, where $\bar{v}_i$ is the state before the transition, and $\bar{v}_f$ is the state after the transition. The state $\bar{v}_f$ is a successor of $\bar{v}_i$ whenever the BDD is satisfied.

Assume that we have computed the BDD representing a subformula f, and wish to compute the BDD representing $\mathbf{EX}f$. This formula is true in a state if and only if there exists a successor of that state which satisfies f. In other words, if the current state is $\bar{v}_i$, there exists a truth assignment to the variables in $\bar{v}_f$ which satisfies $\textsc{Bdd}(f,R)$ such that $R(\bar{v}_i, \bar{v}_f)$ is satisfied. Using boolean quantification, we can express this condition as:

$$\textsc{Bdd}(\mathbf{EX}f, R)(\bar{v}_i) \equiv \exists \bar{v}_f [R(\bar{v}_i, \bar{v}_f) \wedge \textsc{Bdd}(f,R)(\bar{v}_f)].$$

In practice, we first relabel $\textsc{Bdd}(f,R)$ to use the variables of $\bar{v}_f$. Next the logical "and" operation and the existential quantification operation are performed in the same pass over the BDD's. This is done to reduce the storage required for the intermediate results.

Recall that the condition $\mathbf{E}[f\mathbf{U}g]$ means that there is a computaion beginning in the current state in which g is true sometime in the future, and f is true in all the preceding states. This means that either g is true in the current state, or f is true in the current state and there exists a successor state in which $\mathbf{E}[f\mathbf{U}g]$ is true. More formally, it is the least fixed point of condition Z in the expression

$$Z = g \vee [f \wedge \mathbf{EX}Z].$$

This fixed point can be evaluated iteratively, using the BDD representations for f and g. We set Z initially to *false*, then repeatedly evaluate the above expression, using the BDD logical operations and the **EX** procedure described above, until a fixed point is reached. Detecting the fixed point is easy, since testing equivalence of BDD's is a constant time operation. This algorithm can be thought of as finding the set of states which satisfy g, calling it Z, then augmenting this set by adding the set of states which satisfy f and have successors in Z, and repeating this procedure until the set is unchanged.

The formula $\mathbf{EG}f$ states that there exists a computation beginning with the current state in which f is globally true. This means that f is true in the current state, and $\mathbf{EG}f$ is true in some successor state. This condition is the greatest fixed point of Z in the expression

$$Z = f \wedge \mathbf{EX}Z.$$

The BDD representing this fixed point can be computed by setting Z to the BDD constant *true* and repeatedly evaluating the above expression until a fixed point is reached. Intuitively, this is equivalent to beginning with the set of states in which f is true, then removing all those states which have no successors in the set until the set is unchanged.

Finally, in the case of formulas of the form $f \vee g$ or $\neg f$, the logical operations on BDD's can be used to compute BDD's which are true if and only if the formula is true. Since $\mathbf{AX}f$, $\mathbf{A}[f\mathbf{U}g]$ and $\mathbf{AG}f$ can all be rewritten using just the above operators, the above procedure covers the entire logic.

4.1 Fairness Constraints

Next, we consider the issue of *fairness*. In many cases, we are only interested in the correctness along fair computation paths. For example, we may wish to consider only those computations in which some resource that is continuously requested by a process will eventually be granted to the process. This type of property cannot be expressed directly in CTL. In order to handle such properties we must modify the semantics of CTL slightly. A *fairness constraint* can be an arbitrary formula of the logic. A path is said to be *fair* with respect to a set of fairness constraints if each constraint holds *infinitely often* along the path. The path quantifiers in CTL formulas are now restricted to fair paths. In the remainder of this section we describe how to modify the new algorithm to handle fairness constraints. For simplicity, we only consider the case where there is a single fairness constraint B.

Now first reconsider the formula $\mathbf{EG}f$ given a fairness constraint B. This means that there exists a computation beginning with the current state in which f holds globally, *and* B holds infinitely often. The set of such states is the largest set characterized by the following two properties:

1. All of the states satisfy f, and
2. all of the states have a path inside the set to a state satisfying B, of length one or greater.

It is easy to show that if these conditions hold, each state in the set is the beginning of an infinite compute tinal path on which f is always true, and B holds infinitely often. The second condition can be expressed as $\mathbf{EX}(\mathbf{E}[Z\mathbf{U}(Z \wedge B)])$, where Z is the condition that the current state falls within the set. This gives us a characterization of $\mathbf{EG}f$ as the greatest fixed point of Z in the expression

$$Z = f \wedge \mathbf{EX}(\mathbf{E}[Z\mathbf{U}(Z \wedge B)]).$$

This fixed point can be evaluated in the same manner as before. The main difference is that in this case, each time the above expression is evaluated, it causes an **EU** subformula to be evaluated, which itself involves computing a fixed point.

The cases of $\mathbf{EX}f$ and $\mathbf{E}[f\mathbf{U}g]$ under fairness constraints are a bit simpler. The set of all states which are on some fair computation is $Fair = \mathbf{EG}true$, using the above definition of **EG**. Under the fairness constraint, $\mathbf{EX}f$ is just $\mathbf{EX}(f \wedge Fair)$, while $\mathbf{E}[f\,U\,g]$ is $\mathbf{E}[f\mathbf{U}(g \wedge Fair)]$.

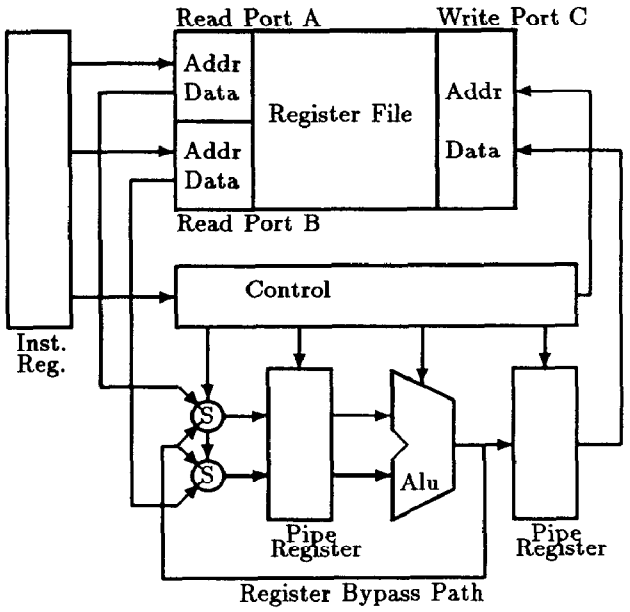

Figure 2: Block diagram of simple pipeline design

Computing fixed points in the above algorithms can be made more efficient in some cases by using the *iterative squaring* technique [6, 7]. A fixed point that would otherwise require n iterations to compute can potentially be computed in $\lceil \lg(n) \rceil$ iterations using this technique.

5 Synchronous Pipeline

As an example, we use a very simple pipeline that performs three-address logical and arithmetic operations on a register file with a three stage pipe. In the first stage, the operands are read from the register file, in the second stage an ALU operation is performed, and in the third stage the result is written back to the register file. The ALU has a register bypass path, which allows the result of an ALU operation to be used immediately as an operand on the next clock cycle, as is typical in RISC instruction pipelines. The inputs to the circuits are an instruction code, containing the register addresses of the source and destination operands, and a STALL signal, which indicates that the instruction stream is stalled. When this occurs, a "no-operation" is propagated through the pipe. A functional block diagram of a typical pipeline is given in Figure 2. A BDD representing the transition relation of this system was extracted from a set of logic equations describing the design. Given a MOS transistor circuit implementing the design, it would also be possible to extract the logic equations using Bryant's method of symbolic simulation [5].

Using the operators of CTL, we can specify the cor-

rect behavior of the pipeline, taking into account the pipe latency. When an instruction is input to the pipeline, it is implicit that that the results of the operation will not affect the register file until three clocks cycles in the future. Likewise, the source operands will be drawn from the results of the previous operation, which will be in the register file two cycles in the future. This observation allows us to specify the correct temporal and functional relationship between instructions and operands in the register file, although space considerations do not allow a specification in temporal logic to be described here. The CTL formulas for these properties will be provided in [6].

Table 1 summarizes the results we obtained in verifying a variety of pipelines of this type. The most complex pipeline we verified was an adder pipeline with four 8-bit registers. It had approximately 5×10^{20}, states, which puts it far outside the range of model checkers like the one reported in [3]. It required a BDD with 79,986 nodes to represent the transition relation, and approximately an hour to verify on a Sun 3 workstation. The most interesting result is that the number of nodes in the transition relation BDD increases *linearly* in the number of bits per register, although the number of states increases exponentially. This results from the structure of communication within the pipeline, and the ordering chosen for the state variables in the BDD. The variables were ordered with the control state first, followed by the all the state bits for the bit 0 slice, then all the state bits for the bit 1 slice, etc. Traversing the BDD from the root to a leaf, there are a fixed number of ways to go from bit slice n to bit slice $n+1$. Each of these paths corresponds to a particular configuration of the control bits and the carry bit out of slice n in the ALU. Consequently, the number of nodes in the BDD grows in simple proportion to the width of the data path.

It is also interesting to note that adding an exclusive-or operation to the addition pipeline roughly doubles the number of nodes in the transition relation characteristic function. This results from the addition of a bit to the control information that must be passed down through the data path levels of the BDD, effectively doubling the number of control states. The complexity of control would therefore seem to be a crucial factor in the size of the BDD representation.

ALU ops	word size	number of registers	BDD size	verification time (secs)
$\oplus$	1 bit	4	2737	9
$\oplus$	2 bits	4	8430	46
$\oplus$	3 bits	4	14123	145
$\oplus$	4 bits	4	19816	306
$\oplus$	8 bits	4	41000	1349
+	1 bit	4	2737	9
+	2 bits	4	10734	45
+	3 bits	4	22276	179
+	4 bits	4	33818	492
+	8 bits	4	79986	3709
$+, \oplus$	2 bits	4	18429	188
$+, \oplus$	3 bits	4	36239	690
$+, \oplus$	4 bits	4	53924	1706

Table 1: Performance of BDD model checking algorithm on simple pipelines

6 Asynchronous Stack

This section describes the verification of an asynchronous stack circuit due to Martin [13]. The circuit uses a variant of the standard 4-phase protocol for communicating with its environment. We verified the circuit by composing it with a "most general" environment that obeys this protocol, and checking that the resulting circuit had no hazards. Dill [11] has shown that this method is adequate to verify safety properties of asynchronous circuits.

The algorithm searches the set of states reachable from the initial state. The circuit is correct if and only if no hazard can occur in any reachable state. If S_n is the set of states reachable in n or fewer transitions, then the states reachable in $n+1$ or fewer transitions is the set of v such that

$$S_n(\bar{v}) \vee \exists \bar{u}[S_n(\bar{u}) \wedge N(\bar{u}, \bar{v})].$$

The algorithm consists of evaluating the above expression iteratively until a fixed point is reached, or until a hazard state is found. A more detailed description of the algorithm can be found in [6].

For asynchronous circuits, the next state relation is often too large to store if it is represented with a single BDD. So, we represent the transition relation as a list of BDD's, one corresponding to each component of the circuit. The next state relation is the disjunction of these BDD's. Each of the BDD's is processed separately as the state space is searched. We also made use of a technique of Coudert, Berthet, and Madre in [10] for simplifying a boolean function under a constraint, in order to reduce the size of the BDD's used in the state space search.

The performance of the BDD-based verifier on an asynchronous stack element is summarized in Table 2. The figure given for the size of the BDD representing the reached state set is the largest for any iteration. This does not in general correspond to the final (and hence largest) set of reached states, since the complexity of the BDD representation is not directly related to the cardinality of the set.

data bits	approx. gate equiv's	BDD size	number of reached states	verification time (secs)
1	30	458	272	20
2	50	865	1632	60
3	70	1735	14696	208
4	90	3101	155024	726
5	100	4774	$\approx 1.5 \times 10^6$	1878
6	120	7968	$\approx 1.5 \times 10^7$	4588
7	140	12051	$\approx 1.5 \times 10^8$	10416

Table 2: Performance of BDD algorithm for asynchronous stack element

7 Conclusions

Our examples show that the state-explosion problem can sometimes be circumvented by using a symbolic representation for state graphs. When the representation captures the right structural uniformities in the graph, it is much smaller than an explicit table of all of the states. The method presented here is not necessarily a replacement for brute-force state-enumeration methods, but an alternative that may work efficiently when the brute force methods fail.

Our method is not especially dependent upon the properties of binary decision diagrams. Any representation of boolean functions that has algorithms for boolean connectives, restriction and equality testing can be used. Although we have concentrated on temporal-logic model checking, BDD's can be used in other formalisms for reasoning about large state transition graphs, such as observational equivalence and automata on infinite sequences [7].

References

[1] S. Bose and A. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In *IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design*, 1989.

[2] S. Bose and A. Fisher. Verifying pipelined hardware using symbolic logic simulation. In *IEEE International Conference on Computer Design*, 1989.

[3] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Trans. Comput.*, C-35(12):1035–1044, 1986.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8), 1986.

[5] R. E. Bryant. Verifying a static ram design by logic simulation. In J. Allen and F. T. Leighton, editors, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*, pages 335–349. MIT Press, 1988.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. Technical report, Carnegie Mellon University, School of Computer Science, 1990. In Preparation.

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, June 1990. To Appear.

[8] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Proceedings of the Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Syst.*, 8(2):244–263, 1986.

[10] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.

[11] D. L. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. In Jonathan Allen and F. Thomson Leighton, editor, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*. MIT Press, 1988.

[12] M. Fujita and H. Fujisawa. Specification, verification, and synthesis on control circuits with propositional temporal logic. In *Ninth International Symposium on Computer Hardware Description Languages and their Applications*. North-Holland, June 1989.

[13] A. J. Martin. A synthesis method for self-timed VLSI circuits. In *Proceedings of the IEEE International Conference on Computer Design*, 1987.