Sequential Circuit Verification Using Symbolic Model Checking

J. R. Burch E. M. Clarke K. L. McMillan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

David L. Dill
Computer Science Laboratory
Stanford University
Stanford, CA 94305

February 18, 1997

Abstract

The temporal logic model checking algorithm developed by Clarke, Emerson, and Sistla [8] is modified to represent a state graph using *binary decision diagrams* (BDD's) [5]. 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 fact 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([14]). 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 ([4], [12], [10]), 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. Developing such techniques has proven to be a very difficult problem. The regularity of data path designs provides some reason to believe that their state graphs, while large, will often have a relatively simple structure. Consequently, it may be possible to find a concise representation that exploits the uniformity of the state space and depends in size more on the inherent complexity of the data path logic than simply the number of states it determines.

In this paper, we show how a technique for reasoning about sequential circuits, called *temporal logic model checking* [7, 8], can modified to represent a state graph using *binary decision diagrams* (BDD's) [5]. 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. One approach 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. This technique has been used by Bryant to verify a MOS memory circuit [5]. 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 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 [9]. 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 [11] describes a verification procedure based on linear temporal logic that uses binary decision diagrams. There are, however, several important differences between Fujita's method and the one described here. First, Fujita's technique does not use model checking. Both the system and it's specification are defined in temporal logic. Second, binary decision diagrams are used only to represent the transition conditions in the automata which are derived from the temporal logic formulas. Each state in the these automata is still explicitly represented. 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. They do not provide empirical results on the algorithm's performance, however. Also, their algorithm does not support fairness constraints [8], so it is of limited use in proving liveness properties.

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 [8]. In this logic each of the usual forward-time operators of linear temporal logic (**G** *globally* or *invariantly*, **F** *sometime in the future*, **X** *nexttime* and **U** *until*) must be directly preceded by a *path quantifier.* The path quantifier can either be an **A** (for all computation paths) or an **E** (for some computation path). Thus, some typical CTL operators are $\mathbf{AG}f$, which will hold in a state provided that f holds at all points (globally) along all possible computation paths starting from that state, and $\mathbf{EF}f$, which will hold in a state provided that there is a computation path such that f holds at some point in the future on the path.

For explaining our verification procedure, it is convenient to express the CTL operators with universal path quantifiers in terms of the operators with existential path quantifiers, taking advantage of the duality between universal and existential quantification. Consequently, in our description of the syntax and semantics of CTL, we specify the existential path quantifiers directly and treat the universal path quantifiers as syntactic abbreviations. Let P be the set of atomic propositions, then:

1. Every atomic proposition p in P is a formula in CTL.
2. If f and g are CTL formulas, then so are $\neg f$, $f \wedge g$, $\mathbf{EX}f$, $\mathbf{E}[f\mathbf{U}g]$ and $\mathbf{EG}f$.

The semantics of a CTL formula is defined with respect to a *Labeled State Transition Graph.* A Labeled State Transition Graph is a 5-tuple $\mathcal{M} = (P, S, L, N, S_0)$ where P is a set of atomic propositions, S is a finite set of states, $L{:}S \rightarrow \mathcal{P}(P)$ (where $\mathcal{P}(P)$ is the powerset of P) is a function labeling each state with a set of atomic propositions, $N \subseteq S \times S$ is a transition relation, and S_0 is the set of initial states. Throughout this paper, for any set R, we say the predicate $R(a, b)$ is true if and only if $\langle a, b\rangle \in R$. This

notation is used to define a *path* as an infinite sequence of states $s_0, s_1, s_2, \ldots$ such that $N(s_i, s_{i+1})$ is true for every i.

The propositional connectives $\neg$ and $\wedge$ have their usual meanings of negation and conjunction. The other propositional operators can be defined in terms of these. $\mathbf{X}$ is the *nexttime* operator. $\mathbf{EX}f$ will be true in a state s of $\mathcal{M}$ if and only if s has a successor s' such that f is true at s'. $\mathbf{U}$ is the *until* operator. $\mathbf{E}[f\mathbf{U}g]$ will be true in a state s of $\mathcal{M}$ if and only if there exists a computation path starting in s and an initial prefix of the path such that g holds at the last state of the prefix and f holds at all other states along the prefix. The operator $\mathbf{G}$ is used to express the *invariance* of some property over time. $\mathbf{EG}f$ will be true at a state s if there is a path starting at s such that f holds at each state on the path. If f is true in state s of structure $\mathcal{M}$, we write $\mathcal{M}, s \models f$. We will identify a CTL formula f with the set $\{s : \mathcal{M}, s \models f\}$ of states that make f true. We also use the following syntactic abbreviations for CTL formulas:

- $\mathbf{AX}f \equiv \neg\mathbf{EX}\neg f$ which means that f holds at all successor states of the current state. (f must hold at the *next* state.)
- $\mathbf{EF}f \equiv \mathbf{E}[\mathbf{true}\mathbf{U}f]$ which means that for some path, there exists a state on the path at which f holds. (f is *possible* in the future.)
- $\mathbf{AF}f \equiv \neg\mathbf{EG}\neg f$ which means that for every path, there exists a state on the path at which f holds. (f is *inevitable* in the future.)
- $\mathbf{AG}f \equiv \neg\mathbf{EF}\neg f$ which means that for every path, at every node on the path f holds. (f holds *globally* or *invariantly* along all paths.)
- $\mathbf{A}[f\mathbf{U}g]) \equiv \neg\mathbf{E}[\neg g\mathbf{U}\neg f \wedge \neg g] \wedge \neg\mathbf{EG}\neg g$ which means that for every path, there exists an initial prefix of the path such that g holds at the last state of the prefix and f holds at all other states along the prefix. (f holds *until* g holds, along all paths.)

There is a program called EMC (Extended Model Checker) that verifies the truth of a formula in a model by using efficient graph-traversal techniques. If the model is represented as a state transition graph, the complexity of the algorithm is linear in the size of the graph and in the length of the formula. The algorithm is quite fast in practice (See [8] and [3] for details) if the behaviors of the circuit being verified are represented as a labeled state transition graph. However, an explosion in the size of the model may occur when the labeled state transition graph is extracted from

a circuit, particularly if the circuit contains many registers or other memory elements. The new model checking algorithm described in this paper was developed to help alleviate this problem. By combining the new algorithm in this paper with decomposition techniques like the one described in [6], it should be possible to handle much larger circuits than was previously the case.

3 Binary Decision Diagrams

We only give a short, intuitive description of boolean decision diagrams in this paper. For a more complete description, see [5].

Figure 3 is an example of a *Binary Decision Diagram (BDD)* representing a boolean function $f(a, b, c, d)$. It can be seen that $f(1, 0, 1, 1) = 1$ as follows: 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 method can be used to completely determine the function represented by a BDD.

A BDD has the additional restriction that a total ordering is placed on the set of variables of the function being represented. For any node b in the diagram with corresponding variable v, all nodes reachable from b must have corresponding variables that are strictly greater than v in this total ordering. This restriction makes it possible to manipulate BDD's much more efficiently than would otherwise be the case. The variable ordering in Figure 3 is $a < b < c < d$. Altering the variable order can have a major impact on the size of the BDD needed to represent a given function.

There are also other restrictions on the form of BDD's, see [5] for details. These result in BDD's having a canonical form. 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 $g(b, c, d) = f(0, b, c, d)$.

The only operations on BDD's that our algorithms require are checking for boolean equivalence, computing boolean connectives, and computing restrictions. All the other operations that are needed can be defined in terms of these. Thus, even though we assume throughout the paper that BDD's are being used, any technique for representing boolean functions can be used

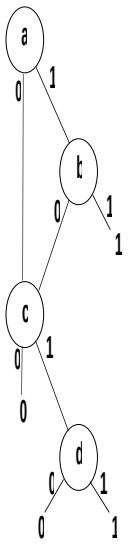

Figure 1: A Binary Decision Diagram

if it provides these three basic operations.

A relation $R(a,b,c,d)$ can be represented by using a BDD to represent its characteristic function. The characteristic function $\mathcal{K}_R(a,b,c,d)$ of a relation $R(a,b,c,d)$ is defined by

$$\mathcal{K}_R(a,b,c,d) = \begin{cases} 1 & \text{if } R(a,b,c,d); \\ 0 & \text{otherwise.} \end{cases}$$

Operations on sets or relations represented in this way can be computed quite easily. For example, the characteristic function of $R_0(a,b,c,d) \cup R_1(a,b,c,d)$ is equal to $\mathcal{K}_{R_0(a,b,c,d)} \vee K_{R_1(a,b,c,d)}$, which can be computed with the algorithms in [5].

It is also possible to quantify over boolean variables when relations are defined in this way. For example, the characteristic function of the relation $\exists a\, R(a,b,c,d)$ is equal to

$$K_{R_0(a,b,c,d)}|_{a=0} \vee K_{R_0(a,b,c,d)}|_{a=1}$$

Universal quantification is computed by replacing the disjunction in the above formula with a conjunction.

4 Representing Transition Graphs Symbolically

Recall that a *Labeled State Transition Graph* is a 5-tuple $\mathcal{M} = (P, S, L, N, S_0)$. When representing a labeled state transition graph $\mathcal{M}$ symbolically, we will assume that $\mathcal{M}$ satisfies the following requirement:

Assumption 1 *For any two states s_1 and s_2 in S, if $L(s_1) = L(s_2)$, then $s_1 = s_2$.*

This assumption causes no loss of generality since extra atomic propositions can be added in order to make $L(s_1) \neq L(s_2)$ for any two distinct states s_1 and s_2. Let V_P be the set of possible truth assignments to the atomic propositions in P. We write $\bar{u}$, $\bar{v}$, $\bar{w}$, etc., to denote elements of V_P. Thus, $\bar{u}(p) = 1$ if the atomic proposition p is true in the truth assignment $\bar{u}$; otherwise, $\bar{u}(p) = 0$. Any $\mathcal{M}$ satisfying the above assumption is isomorphic to a labeled state transition graph with a subset of V_P as its set of states. This isomorphism maps a state s to the $\bar{u} \in V_P$ such that

$$L(\bar{u}) = \{p \in P : \bar{u}(p) = 1\}$$

Thus, there is no loss of generality in

Assumption 2 *$\mathcal{M} = (P, S, L, N, S_0)$ where $S = V_P$ and*

$$L(\bar{u}) = \{p \in P : \bar{u}(p) = 1\}.$$

Satisfying this may require adding additional states to the structure, but these states will be unreachable and will not affect the truth of any CTL formulas. We need not represent the labeling function L explicitly since it will always have the form described in Assumption 2. In this context, we will often refer to atomic propositions as *state variables.*

Given Assumption 2, the set of states of a labeled state transition graph can be represented by a relation $R(\bar{u})$. The number of variables in the resulting BDD is equal to $|P|$. The set of initial states S_0 can be represented with a BDD in this way. Similarly, the next state relation $N(\bar{u}, \bar{v})$ of a labeled state transition graph is represented by a BDD with $2(|P|)$ variables.

4.1 Computing the Set of Reachable States

The set C_n of states reachable in n or fewer steps is defined inductively by:

$$\begin{aligned} C_0(\bar{u}) &= S_0(\bar{u}) \\ C_{n+1}(\bar{u}) &= S_0(\bar{u}) \vee \exists \bar{v} \, [C_n(\bar{v}) \wedge N(\bar{v}, \bar{u})] \end{aligned}$$

If C_* is the set of reachable states, then there exists an n such that $C_* = C_n$. In fact, $C_* = C_m$ for all $m \geq n$. This fact gives an algorithm for computing C_*: compute $C_0, C_1, \ldots, C_{n+1}$ until $C_{n+1} = C_n$. Then, $C_* = C_n$.

5 Iterative Squaring

Computing the set of reachable states of a labeled state transition graph can viewed as finding the least fixed point of the predicate transformer F such that

$$F(Q) = S_0(\bar{v}) \vee \exists \bar{u}\,[Q(\bar{u}) \wedge N(\bar{u}, \bar{v})].$$

The algorithm described above for computing the reachable states computes $F(\emptyset), F(F(\emptyset)), \ldots, F^n(\emptyset), \ldots$ until a fixed point is reached (we use superscript n to denote repeated application). We call this technique for computing a fixed point *direct iteration.*

The predicate transformer F described above has the property that

$$F(Q) = R_0(\bar{v}) \vee \exists \bar{u}[Q(\bar{u}) \wedge R_1(\bar{u}, \bar{v})], \tag{1}$$

where

$$R_0 = S_0 \text{ and } R_1 = N.$$

It is easy to show that any relational predicate transformer of the form in Equation (1) has both a least and a greatest fixed point. Computing these fixed points can be made more efficient in some cases by making use of the following two facts.

Proposition 3 *If F is predicate transformer of the form in Equation (1), then there exists an n such that $\forall m\,[m \geq n \Rightarrow F^m = F^n]$.*

We write F^* to denote the F^n described in the previous proposition.

Proposition 4 *If F is predicate transformer of the form in Equation (1), then $F^2(Q) = R_0(\bar{v}) \vee \exists \bar{u}\,\exists \bar{w}\,[Q(\bar{u}) \wedge R_1(\bar{u}, \bar{w}) \wedge R_1(\bar{w}, \bar{v})]$. Therefore F^2 is also of the form in Equation (1).*

A fixed point of a predicate transformer F of the form in Equation (1) can be computed in two steps. First compute F^* by computing

$$F, F^2, F^4, \ldots, F^{2^n}, \ldots$$

until $F^{2^{n+1}} = F^{2^n}$. Then, the least fixed point is equal to $F^*(\emptyset)$ and the greatest fixed point is equal to $F^*(U)$ where U is the universal set. In this

paper, the universal set will always be V_P, which is the set of states according to Assumption 2. We call this technique for computing fixed points *iterative squaring*. Iterative squaring can make the number of iterations necessary to reach a fixed point exponentially smaller than in direct iteration. However, the iterative squaring can be impractical if the BDD's needed to represent the intermediate computations become too large.

6 The Symbolic Model Checking Algorithm

The symbolic model checking algorithm works inductively over the structure of CTL formulas. Computing the boolean connectives is simple with BDD's, so we need only consider CTL formulas of the form $\mathbf{EX}f$, $\mathbf{EG}f$, and $\mathbf{E}[f\mathbf{U}g]$. The first case is straightforward,

$$\mathbf{EX}f = \exists\, \bar{v}[f(\bar{v}) \wedge N(\bar{u}, \bar{v})].$$

The algorithms for **EG** and **EU** are based on a characterization of the CTL operators as fixed points of predicate transformers. An early, inefficient CTL model checking algorithm [7] was also based on this characterization, but was eventually replaced by a more efficient one using depth-first search. The fixed point characterization for **EG** is given by

$$\mathbf{EG}f = f \wedge \mathbf{EX}(\mathbf{EG}f).$$

The intuition for this result should be clear: For every state s, there is a path starting at s along which f holds globally if and only if f holds at s and s has a successor s' such that there is a path starting at s' along which f holds globally.

The above fixed point characterization describes a predicate transformer F of the form in Equation (1), where $R_0 = \emptyset$ and

$$R_1(\bar{u}, \bar{v}) = f(\bar{v}) \wedge N(\bar{v}, \bar{u}).$$

We would like to compute the greatest fixed point of F. As described earlier, this can be done with either the direct iteration method or the iterative squaring method.

The fixed point characterization for **EU** is similar to the one for **EG**:

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

Starting at some state s, there will be a path such that $f\mathbf{U}g$ holds if and only if g holds at s or f holds at s and s has a successor s' such that that there is a path starting at s' along which $f\mathbf{U}g$ holds. The resulting predicate transformer is of the form in Equation (1) with R_1 the same as in the **EG** case and $R_0 = g$. However, this time $\mathbf{E}[f\mathbf{U}g]$ is the *least fixed point* of the corresponding predicate transformer rather than the greatest. Again, this fixed point can be computed using either direct iteration or iterative squaring.

6.1 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. The model checker will initially prompt the user for a series of *fairness constraints*. Each 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, which we treat as a set of states. As in the previous discussion, we show how to handle **EX**, **EU**, and **EG**.

First we describe how to compute the set $Good(B)$, which is the set of states that lie on a fair path under the fairness constraint B. Let

$$Ancestor(f) = \{s \mid s \models \mathbf{EF}f\}.$$

$Ancestor(f)$ is the set of states from which a state satisfying f is reachable. Let $ProperAncestor(f) = \mathbf{EX}Ancestor(f)$. $ProperAncestor(f)$ is the set of states from which a state satisfying f is reachable by a path of length at least one. Given f, we can compute $ProperAncestor(f)$ using the previous model checking algorithm. Let $Loop(B)$ be the greatest fixed point of the predicate transformer F where

$$F(Q) = B \wedge ProperAncestor(Q).$$

A state is in $Loop(B)$ if it is in B and it lies infinitely often on some fair path under the fairness constraint B. The proof of this result will be given in the full version of the paper. $Loop(B)$ can be computed either by direct iteration or by iterative squaring. Let $Good(B) = Ancestor(Loop(B))$. $Good(B)$ is the set of states from which there exist infinite B-fair paths.

To check $\mathbf{EX}f$ under the fairness constraint B, we simply check $\mathbf{EX}(f \wedge Good(B))$. To check $\mathbf{E}[f\mathbf{U}g]$ under the fairness constraint B, we simply check $\mathbf{E}[f\mathbf{U}(g \wedge Good(B))]$. Thus, the problem of checking $\mathbf{EX}f$ and $\mathbf{E}[f\mathbf{U}g]$ with fairness constraints is reduced to the problem of model checking without fairness contraints, and the previously described model checking algorithm can be used.

The **EG** case is more difficult. Let $Ancestor[f](g) = \mathbf{E}[f\mathbf{U}(f \wedge g)]$. $Ancestor[f](g)$ is the set of states from which a state satisfying g is reachable by a path along which f always holds. Let $ProperAncestor[f](g) = f \wedge \mathbf{EX}Ancestor[f](g)$. Also, let $Loop[f](B)$ be the greatest fixed point of the predicate transformer F where

$$F(Q) = B \wedge ProperAncestor[f](Q).$$

A state is in $Loop[f](B)$ if is in B and it lies infinitely often on some fair path (under the fairness constraint B) on which f always holds. The proof of this result will also be given in the full paper. Computing $\mathbf{EG}f$ under the fairness constraint B requires computing

$$Ancestor[f](Loop[f](B)),$$

which can also be done using either direct iteration or iterative squaring.

7 Empirical Results

The Boolean decision graph method for testing boolean satisfiability is only efficient in a heuristic sense. The problem is, of course, NP-complete in general; the only claim that is made for BDD's is that they perform well for certain useful classes of boolean functions. Likewise, The BDD method for representing state sets in the CTL model checking problem is only of heuristic value, and does not improve the asymptotic complexity of model checking. Therefore, in order to evaluate the method, we need empirical results showing the performance of the method for some problems of practical interest. We have examined two classes of digital circuits in evaluating

the method empirically. The first is a class of simple synchronous pipelines, which include data path as well as control circuitry. The number of states in these systems is far too large to apply traditional model checking techniques, but we have obtained very encouraging results using the BDD method.

The second class of circuits are asynchronous designs with data paths. Convergence of the fixed point expressions in these systems generally requires a much larger number of steps, since a large number of independent asynchronous transitions may be required to complete operations which are synchronized on a single clock transition in a synchronous design. The results on the performance of the BDD method for these circuits are more ambiguous than those for the synchronous pipelines; it is not yet clear to us to what degree the BDD method is applicable to this kind of circuit.

7.1 Synchronous pipelines

The circuits we have used as examples of this category are very simple pipelines that perform three-address logical and arithmetic operations on a register file. The complete state of the register file and pipe registers are modeled. The pipelines have three stages. 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.

Since vectors of boolean values are used to represent binary numbers in these designs, it will be useful to introduce some notation for vectors of propositions in logical formulas. First, we define the standard logical and modal operators to operate on vectors of propositions in a component-wise manner. For example,

$$\begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix} \wedge \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_n \end{bmatrix} \equiv \begin{bmatrix} p_1 \wedge q_1 \\ p_2 \wedge q_2 \\ \vdots \\ p_n \wedge q_n \end{bmatrix}$$

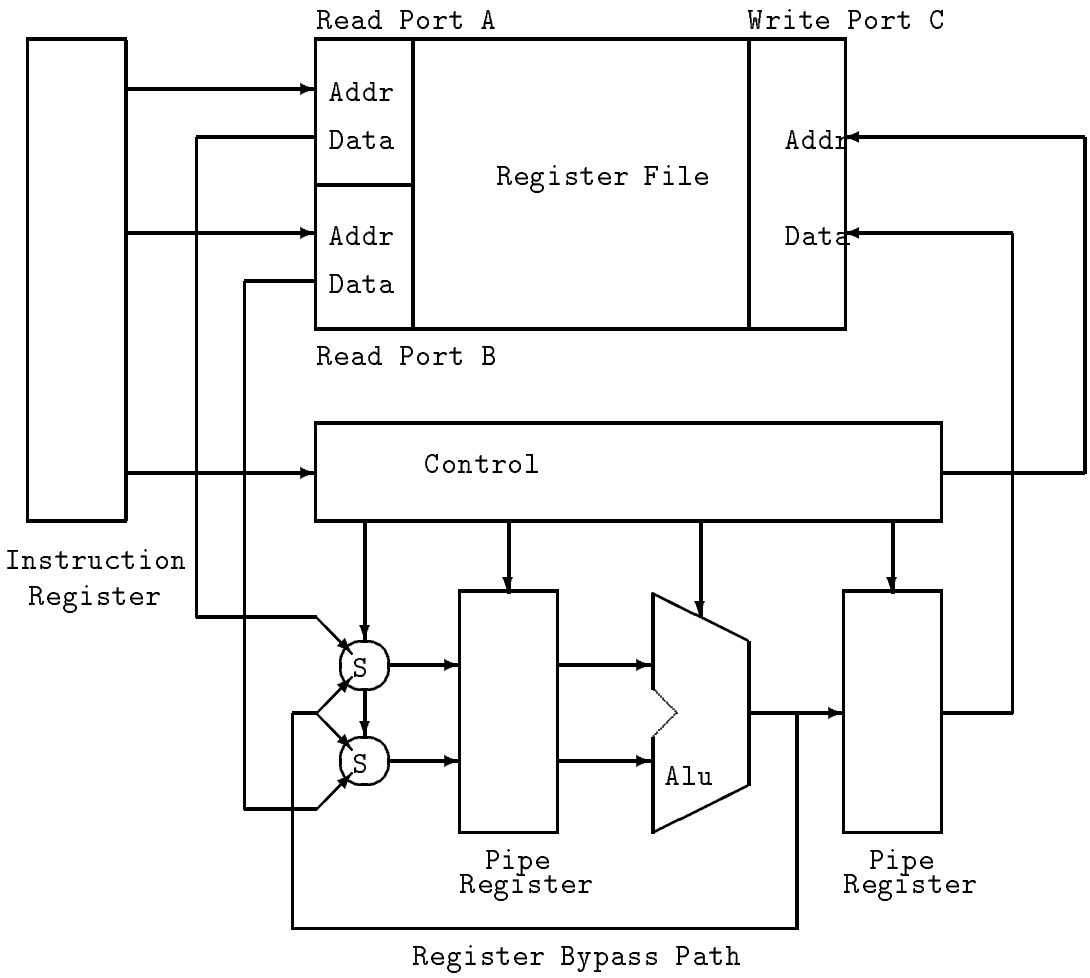

Figure 2: Block diagram of simple pipeline design

and

$$F \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix} \equiv \begin{bmatrix} Fp_1 \\ Fp_2 \\ \vdots \\ Fp_n \end{bmatrix}$$

In order to deal with the register file, it is also useful to define arrays of propositions (vectors of vectors) and a function $select(v_1, v_2)$ which, given vectors v_1 and v_2, returns the element of v_1 indexed by the *binary number* represented by v_2. [1]

The latency in the example pipelines is three clock cycles. For this reason, the specification of the pipeline cannot be given in a straightforward manner using simply pre- and post- conditions on operations.[2] We can, however, use temporal operators and the above notation to specify the behavior of the pipeline, taking into account the pipe latency. When we specify a register transfer level operation for the pipeline, it is understood that the results of the operation will not affect the register file until three clocks cycles in the future, and the inputs to the operation correspond to the state of the register file two clock cycles in the future. The state of the register file n clock cycles in the future can be expressed using the (vector) modal operator "$\mathbf{X}$", as $\mathbf{X}^n\mathbf{R}$. Thus, taking into account the pipe latency, an RTL specification such as $\mathbf{R_c} \leftarrow \mathbf{R_a} \oplus \mathbf{R_b}$ can be expressed as a temporal formula in the following way:

$$select(\mathbf{X}^3\mathbf{R}, \mathbf{c}) = select(\mathbf{X}^2\mathbf{R}, \mathbf{a}) \oplus select(\mathbf{X}^2\mathbf{R}, \mathbf{b})$$

where **a, b** and **c** are each bit-fields in the operation code. As similar formulas can be derived for other RTL expressions, we will write RTL syntax in our specifications, with the understanding that it is to be interpreted in the above (temporal) way. Since $\mathbf{X}^n p$ is a path formula and not a state formula, it can't be evaluated directly by the CTL model checker (which can only evaluate state formulas). We can show, however, that the state of the register file $\mathbf{R}$ two or three clock cycles in the future is uniquely determined by the current state of the system. We can show this by automatically

[1] Note that *select* can be written as a boolean function, in much the same way one might implement a multiplexer using logic gates. It is implemented as a macro in the model checker.

[2] Bose and Fisher[2] describe a method of mapping pipelined systems onto abstract systems for which such a simple specification can be made, but this mapping is not constructed automatically.

checking the CTL formulas

$$\mathbf{AG}((\mathbf{EX})^2\mathbf{R} \equiv (\mathbf{AX})^2\mathbf{R})$$

and

$$\mathbf{AG}(\mathbf{EX})^3\mathbf{R} \equiv (\mathbf{AX})^3\mathbf{R})$$

If these formulas are true, we can substitute the state formula $(\mathbf{EX})^2\mathbf{R}$ for the path formula $\mathbf{X}^2\mathbf{R}$, since the two are equivalent. Likewise, we can substitute $(\mathbf{EX})^3\mathbf{R}$ for $\mathbf{X}^3\mathbf{R}$.

Using the above temporal interpretation for RTL specifications, we can write the specification for our simplest pipeline (which has only an exclusive-or instruction) as follows:

$$\mathbf{AG}\neg STALL \Rightarrow (\mathbf{R_c} \leftarrow \mathbf{R_a} \oplus \mathbf{R_b}) \tag{2}$$

and

$$\mathbf{AG}\forall \mathbf{c}'(\mathbf{c} \neq \mathbf{c}' \vee STALL \Rightarrow (\mathbf{R_{c'}} \leftarrow \mathbf{R_{c'}}))$$

The latter formula makes explicit the assumptions that non-destination registers don't change, and that if a stall occurs, no registers change.

Table 1 summarizes the results we obtained in verifying a variety of pipelines of this type. We varied the number of bits per register, and the type of operation(s) performed by the ALU, to see how these affected the size of the BDD used to represent the transition relation, the total execution time required to check formula 2 and the total storage used. The most complex pipeline we verified had approximately 5×10^{20}, states, which puts it far outside the range of model checkers like the one reported in [4]. It required a BDD with 42000 nodes to represent the transition relation, and approximately 22 minutes on a Sun 3 workstation to verify. The most interesting result is that the number of nodes in the transition relation BDD increases *linearly* in the number of bits per register. This is somewhat surprising on it's face, since the number of states increases exponentially in the number of bits, as does the complexity of boolean satisfiability (as far as we know). On consideration, however, this is perhaps not very surprising. Intuitively, the complexity of the BDD is a function of how much information must be remembered as one passes from one layer of the BDD to the next (ie., from one variable to the next). In the pipeline examples, the information stored from one "bit slice" of the data path to the next is rather small; it amounts to the state of the control bits plus at most the value of the ALU "carry" bit.

ALU ops	word size	number of registers	size of transition relation BDD (nodes)	verification time (secs)
$\oplus$	1 bit	4	2737	9.0
$\oplus$	2 bits	4	8430	46.7
$\oplus$	3 bits	4	14123	145.8
$\oplus$	4 bits	4	19816	306.5
$\oplus$	8 bits	4	41000	1349.0
+	1 bit	4	2737	9.0
+	2 bits	4	10734	45.4
+	3 bits	4	22276	179.5
+	4 bits	4	33818	492.7
+	8 bits	4	79986	
+, $\oplus$	2 bits	4	18429	188.3
+, $\oplus$	3 bits	4	36239	690.7
+, $\oplus$	4 bits	4	53924	1706.0

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

In particular, this amount of information does not increase as one increases the number of bits, so the BDD becomes deeper, but no "wider".

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 fact the an additional bit has been added 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. In addition, if ALU were able to perform a multiply operation, a barrel shift, or some other complex operation which has more than a constant amount of information passing from one bit position to the next, then the size of the BDD representation would quickly become unmanageable. There is some reason to believe, however, that product decomposition methods for finite automata can be used to derive a concise composite BDD representation in these cases.

7.2 Verifying asynchronous and self-timed circuits

The synchronous pipeline experiments show that it is in fact possible to exploit the regularity of some data paths to construct a compact representa-

tion for their state space. Will the same effect be observed for asynchronous circuits? Since the behavior of asynchronous and self-timed circuits is considerably less ordered that that of synchronous circuits, we should expect the complexity of verifying them to be greater. We observed in the previous section that the number of control states had an important impact on the tractability of representing the state space. In self-timed circuits, the number of global control states is generally quite high, due to the loose synchronization between components. On the other hand, a certain kind of regularity can be said to exist in the state space, since many of the possible transitions are mutually commutative — the action of one transition does not affect the enabling conditions of another. A certain symmetry can thus be said to exist in the state graph, since commuting sets of transitions form "hypercubes", which when traversed in a breadth-first manner, produce characteristic functions which are symmetric in their input variables. Further, the BDD representation is known to be compact for symmetric functions. The question is, will this effect compensate for the inherently high number of control states in asynchronous circuits? To test this idea, we applied BDD-based methods to checking hazard freeness in a speed-independent stack element design due to [13].

We take a slightly different approach to asynchronous circuits (as opposed to synchronous circuits) because of a phenomenon we observed in our experiments that led to a more efficient method. The CTL model checking procedure we used for synchronous circuits effectively begins with a set of states and expands that set by backward chaining though the state graph until a fixed point is reached (this is the case for EU type formulas, which are most commonly used for specifying safety properties). This backward-chaining approach provided exactly the result we wanted for specifying the synchronous pipelines, since we we wanted to specify the set of states which could reach a state n steps in the future where a given register bit had a value 1. We discovered, however, that for *asynchronous* circuits, expanding from the set of initial states by *forward* chaining progressed much more rapidly than backward chaining from an arbitrary set of states representing some failure condition. The forward chaining algorithm can be characterized by a fixed point expression in much the same way as the CTL operators discussed in section 6. The set $\mathbf{fc}(S)$ of all states reachable by forward chaining from the set S is the least fixed point of the predicate transformer

$$F(Q) = S(\bar{v}) \vee \exists \bar{u}[Q(\bar{u}) \wedge N(\bar{u}, \bar{v})].$$

This forward-chaining operator is not appropriate to the synchronous pipelines,

because, while the current state of the pipe uniquely determines the eventual future contents of the register file, it does not determine the past state of the operation code register. In order to answer the question "what is the set of states in which a given bit of a source operand address was true two clock cycles in the past?", we would have to add state to the model to preserve this information.

For asynchronous circuits, however, the sets of states obtained by forward chaining from the initial states may have a particularly regular structure. We will posit some reasons for this phenomenon in the next section. Here, we simply observe that where a forward-chaining operator is applicable, we prefer it to a backward-chaining operator for specifying asynchronous circuits. In the case of hazard checking, we first characterize the set of states H in which hazard transitions are possible (see [10] for a discussion of these "auto-failures" in asynchronous circuits) and then check the formula

$$\mathbf{fc}(S_0) \Rightarrow \neg H$$

In other words, "If a state is reachable from the initial state, then it is hazard free." In effect, evaluating this formula results in a breadth-first forward search of the state space.

Another "optimization" which we have found useful for dealing with asynchronous circuits is to represent the transition relation as a list of characteristic functions, one corresponding to each logic element in the circuit. Since the transitions of each element occur asynchronously, and there is no overall structure or regularity in the organization of the elements, this is a more compact representation than the BDD obtained for the union of all of the separate transition relations.

The performance of the BDD-based verifier on asynchronous stack element is summarized in Table 2. The figure given for the size of the BDD representing the reached state set is the 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. The table also gives the number of states reached, and total execution time as a function of the number of data bits. Note that the number of reached states grows by roughly a factor of 10 for each additional data bit, while the number of nodes grow by a factor less than two, and the execution time by a factor between 2 and 3. This is an encouraging result, in that it allows us to check a system with many more states than was previously possibly, and it lends some validity to the

data bits	approx. gate equivalents	depth of search	largest BDD representing reached-state set	number of reached states	verification time (secs)
1	30	44	458	272	20.5
2	50	57	865	1632	60.6
3	70	75	1735	14696	208.8
4	90	93	3101	155024	726.3
5	100	111	4774	$\approx 1.5 \times 10^6$	1878.0
6	120	129	7968	$\approx 1.5 \times 10^7$	4588.7
7	140	147	12051	$\approx 1.5 \times 10^8$	10416.1

Table 2: Performance of BDD algorithm for asynchronous stack element

conjecture that regularity exists in the state graph of asynchronous circuits. Nonetheless, in this case the BDD algorithm has not given us polynomial complexity in the number of data bits. It has only reduced the base of the exponential. It is possible that the exponential complexity derives from the additional control state that is introduced as the completion tree (which synchronizes the individual data bits of the stack element) grows in complexity. In any case, it would seem that more powerful methods are needed to verify circuits of this type with a large number of data bits.

7.3 Frontier set simplification

In the verification of the stack element, we made use of a technique of Coudert, Berthet, and Madre in [9] for simplifying the representation of the set of states on the "search frontier" (ie., the set of states reached but not yet expanded). This set is the input to the next iteration of the fixed point algorithm, so there is some interest in making its representation as compact as possible. The correctness of the search algorithm is not affected by whether or not the set of reached states passed to the next iteration of the algorithm does or does not contain states which have been previously expanded. Using a heap, or other traditional representation for the reached state set, it is important not to re-expand any states that have been previously expanded. However, since the complexity of a BDD is not directly related to the number of states it represents, it is often advantageous to re-expand some states if this will reduce the size of the BDD representing the set of states to be expanded. Coudert, Berthet, and Madre describe a method for simplifying the "frontier set" according to this principle.

We have found that in the case of the stack element, this method reduces the number of nodes in the frontier set BDD by a factor of 2 to 20, depending on the depth of the search. Figure 7.3 shows a graph of the size of the BDD representing the reached states and the simplified frontier set BDD as a function of the number of iterations in the search, for the 6-bit stack element. This graph shows an effect that we have observed a number of times in applying breadth-first search to asynchronous circuits. The graph has a small number of peaks which occur at evenly spaced intervals. We conjecture that these peaks occur at search depths where there is the greatest disparity in the progress of individual elements of the circuits. Completion of a given operation generally occurs after a fixed or nearly fixed number of transitions, even though the order of those transitions is highly arbitrary. These situations correspond to troughs in the graph, where the set of reached states is highly regular, and the set of frontier states is actually very small. This phenomenon may in part explain why a forward-chaining search from the initial state is so much more efficient than a backward-chaining search from some arbitrary set of states representing a failure condition. Even at the peaks of the graph, the fact that the reached states at each iteration are the result of a fixed number of mostly commutative transitions may lend a symmetry to the characteristic function which results in a more compact representation.

8 Conclusions

As our examples show, 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 choice of symbolic representation requires balancing between the expressive power of the representation and the existence of good algorithms for manipulating it.

In our examples, we used binary decision graphs as the symbolic representation. Results so far indicate that this representation works well much, but not all of the time. Thus, the method 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 graphs. Any representations of boolean functions that supports boolean operations and for which there are good simplification algorithms

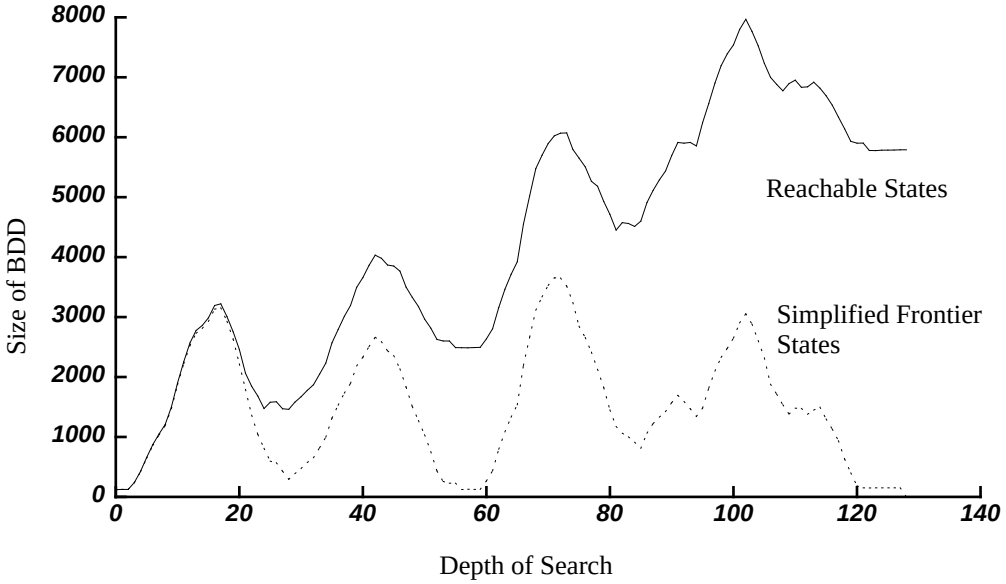

Figure 3: BDD size as a function of Search Depth for the 6-bit Stack.

is a candidate as an internal representation. This is fortunate; because of the importance of boolean functions in CAD for digital systems, a great deal of effort will continue to go into finding better representations and algorithms for manipulating boolean functions. As better representations are developed, they can easily be plugged into our framework to give better verification methods as well.

Although we have concentrated on temporal-logic model checking, the symbolic state graphs (and specifically binary decision graphs) can be used in other formalisms for reasoning about sequential and concurrent behavior, such as propositional linear temporal logic and automata on infinite sequences.

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*, Houthalen, Belgium, 1989. To Appear.

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

[3] M. C. Browne. An improved algorithm for automatic verification of finite state machines using temporal logic. In *Proceedings of the Conference on Logic in Computer Science*, June 1986.

[4] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12), December 1986.

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[6] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of Fourth Symposium on Logic in Computer Science*, 1989.

[7] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Proc. of the Workshop on Logic of Programs*, Springer-Verlag, Yorktown Heights, NY, 1981.

[8] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[9] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, Springer-Verlag, Grenoble, France, 1989. To Appear.

[10] David 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.

[11] M. Fujita and H. Fujisawa. Specification, verification, and synthesis on control circuits with propositional temporal logic. In *Ninth IFIP Symposium on Computer Hardware Description Languages*, pages 265–279, Elsevier Science Publishers, Washington, DC, USA, 1989.

[12] R. P. Kurshan. *Testing Containment of ω-Regular Languages*. Technical Report 1121-861010-33-TM, Bell Laboratories, 1986.

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

[14] Jorgen Staunstrup, Stephen Garland, and John Guttag. Compositional verification of vlsi circuits. In *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, Springer-Verlag, Grenoble, France, 1989. To Appear.