Verification of Speed-Independent Data-Path Circuits

David T. Weih
Mark R. Greenstreet
Department of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z4
Canada

Abstract

This paper demonstrates that verification techniques developed for relatively large, synchronous circuits can be applied to speed-independent, self-timed circuits. We introduce *local formulas* which provide a natural way to specify the input/output behaviour of data-path circuits. The validity of a local formula is independent of the order in which the operations occur in a speed-independent circuit. We demonstrate our approach with the verification of two designs: a FIFO, and a vector multiplier chip.

1 Introduction

Synchronous and asynchronous designs differ primarily by the timing models used to describe their behaviour. In a pure synchronous design, computation is performed by combinational logic, and storage is performed by clocked latches. In this framework, the state of the latches after a clock event is a deterministic function of the values held in the latches before the clock event. Specialised verification methods, such as symbolic trajectory evaluation [Seg93], exploit the fact that synchronous designs have deterministic next state functions which allow a symbolic "next state" operator to be implemented efficiently. This allows large synchronous designs to be formally verified with a high degree of automation [Bea93, Dar94, AS95].

Asynchronous designs, on the other hand, don't use a clock to determine the timing of state changing operations. Because the timing relations between circuit components cannot be completely determined, asynchronous designs are usually modeled using non-deterministic next-state relations. Symbolic model checkers [BCo+94] have been used to verify small designs;

however, the state-space explosion problem prevents fully automated verification on the same scale that has been achieved for synchronous designs.

This paper introduces a framework in which speed-independent circuits without output choice can be verified using synchronous models. The key observation is that the sequence of operations performed by each component depends only on the initial state at the sequence of input values applied by the environment. This motivates specifying the desired behaviour in terms of input and output sequences which is a natural approach for data path circuits, and can be readily verified using symbolic trajectory evaluation. By examining a specific ordering of circuit operations, the circuit can be verified using tools developed for synchronous designs.

By excluding circuits with output choice, we exclude designs that employ arbiters. We note that many published asynchronous designs meet this requirement. We verify a vector multiplier from [SS93] to demonstrate the practicality of our approach. The Caltech Asynchronous Processor [MB+89] and Williams' self-timed divider [Wil91] are other well-known designs that do not use arbiters. A notable exception is the Sproull Counter Flow Pipeline Processor [SSM94] in which arbiters are used to coordinate the transfer of instructions and data between adjacent pipeline stages.

2 Properties of Speed-Independent Circuits

This section presents the properties of speed-independent circuits that we use. The key observation is that the sequence of operations performed by each component in a speed-independent circuit depends only on the initial state of the circuit and the sequence of values applied by the circuit's environment. This observation, motivates using specifications that describe the sequences of inputs and outputs of specific components. The design of a simple FIFO is used as an example throughout this section.

2.1 Specifications

We verify the correct operation of a circuit for all possible initial states, inputs and relative timings of circuit components. To model all possible inputs, the circuit must be described with its environment. Typically, the description of the environment describes the handshaking protocol and allows the environment to apply arbitrary data values. Our verification then shows that the circuit produces the correct results for all possible data values.

We view the circuit and the environment as collections of components (e.g. gates, latches, C-elements). Each signal is connected to the output of exactly one component of either the circuit or the environment. A circuit is **speed-independent** if no assumptions are made about the relative delays of components, and once a component is enabled to change its output(s), it remains enabled to perform that action until it has performed some action.[1] A component has **output choice** if it can be enabled to perform one of two or more mutually exclusive actions. A component has **stable inputs** if once it is enabled, the values of its input nodes remain unchanged until it performs an action. We consider designs where the circuit and environment are speed-independent, circuit components do not have output choice, and environment components have stable inputs. Such a design is called an **eligible** design.

A FIFO in a generic environment

in.t	*in.f*	*in*
L	L	E: empty
L	H	F: false
H	L	T: true
H	H	U: illegal

Dual-Rail Code

Figure 1: A self-timed FIFO and its environment

As an example, figure 1 shows a FIFO with its environment. The data source module provides data to be input to the FIFO using a dual-rail code and a four-phase handshake protocol. When the request signal from the FIFO, *r0*, is high, the source is enabled to send a value by changing in from empty to either true or false. When this value has been inserted into the FIFO, the FIFO lowers *r0*, and the source responds by returning in to the empty value. The FIFO responds to this empty value by raising *r0* completing the four-phase handshake. In an analogous manner, the data sink module consumes values output by the FIFO.

Our specification for the FIFO requires that the sequence of values output by the FIFO be the same as the sequence applied by the source. If c is a component with stable inputs, and v is an input or output of c, we say v_c^k is a **local literal** which represents the value of v after the k^{th} operation of c. Thus, the sequence of values output by the source can be written as in_{src}^1, in_{src}^2, ... To specify that the input and output sequences are the same, we

[1] Our analysis of speed-independence is similar to the analysis of semi-modular circuits in [MB59]

write

$$\forall i \in \{1 \ldots +\infty\}. out^i_{\text{snk}} = in^i_{\text{src}} \tag{1}$$

A **local formula** is a boolean valued expression whose literals are local literals. A local formula is bounded if there exists some integer m such that for every local literal, v^k_c in the formula, $k \leq m$. Equation 1 is an unbounded local formula. Because local formulas describe the input/output behaviour of a circuit, they are often written using only interface signals and environment components. This means that the specification is independent of the implementation, and the same specification can be used for the verification of several different implementations.

The specification for a FIFO in equation 1 assumes that the FIFO is initially empty. A complete specification requires a local formula and an initial state predicate. An **initial state predicate** is a boolean valued expression whose literals are the signals of the circuit and environment. Typically, the circuit to be verified has internal state holding components, and the initial state predicate must describe the state of these components. Thus, the initial state predicate typically includes clauses that are specific to the implementation being verified.

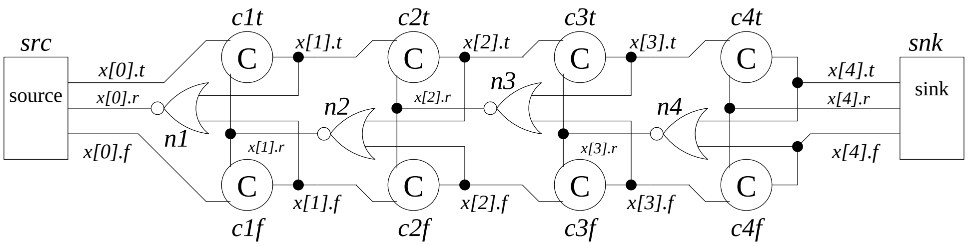

Figure 2: Implementation of a four-stage FIFO

Figure 2 shows an implementation of a four-stage FIFO. The dual-rail pair *(x[0].t, x[0].f)* corresponds to the signal *in* from figure 2. Likewise, *x[0].r* corresponds to *r0*, *(x[4].t, x[4].f)* corresponds to *out*, and *x[4].r* corresponds to *r1*. With these changes, the specification from equation 1 becomes

$$\forall i \in \{1 \ldots +\infty\}.(x[4].t^i_{\text{snk}}, x[4].f^i_{\text{snk}}) = (x[0].t^i_{\text{src}}, x[0].f^i_{\text{src}}) \tag{2}$$

If the FIFO is initially empty, then for each stage i, *x[i].t* and *x[i].f* are false. These observations lead to the initial state predicate below:

$$\bigwedge_{i=0}^{4} \neg x[i].t \wedge \neg x[i].f \tag{3}$$

2.2 Traces

The behaviour of a design is given by a sequence of states. This sequence is called a **trace**. If t is a trace, we write $t(k)$ to denote the k^{th} state of t. The first state in a trace must satisfy the initial state predicate of the design. Each pair of consecutive states in a trace corresponds to an action taken by exactly one component of either the circuit or the environment. A design satisfies a local formula if every trace of the design satisfies the formula.

A circuit may give rise to many traces. There are three causes for this non-determinism:

1. Different traces may start in different initial states. For example, the initial state predicate for the FIFO (equation 3, does not specify the initial value of *x[i].r* for $0 \leq i \leq 4$. Thus, this circuit has thirty-two possible initial states.

2. From the same state, an environment component may be enabled to output one of several different values. For example, when *x[0].r* is true, the *src* component may raise either *x[1].t* or *x[1].f* to output either a true or a false value respectively.

3. Two or more components may be enabled in the same state, and any enabled component may effect the next state change. For example, after a value is copied from the source to the first stage of the FIFO, the next action might be to copy this value to the second stage, to lower *x[0].r*, or to propagate a value or request in some other stage of the FIFO. It is this non-deterministic change of circuit state that distinguishes self-timed design from traditional, synchronous designs.

Our verification method handles the first two causes of non-determinism by representing the initial state and inputs from the environment symbolically. In this way, traces arising from all possible initial states and all possible inputs are represented by a single boolean formula. In the remainder of this section, we show that the value of a local formula is independent of the order in which components perform their actions.

2.3 Characterising traces

Let t and u be traces of an eligible design. If t and u satisfy the same bounded local formulas, then we say that t and u are **equivalent**. Equivalent traces are those which differ only in the order in which their components perform their actions. Formalising and proving this claim requires a few definitions given below.

Let t be a trace, c be a component, and k be an integer such that c is enabled to perform an action in state $t(k)$. Because the design is speed-independent, c will remain enabled until it performs an action. However, we could imagine a trace in which c remains enabled forever and never performs an action. Allowing concurrently enabled components to perform their actions in any order models the unspecified delays in the design. We assume that these delays are bounded. Therefore, we assume that every time a component becomes enabled, it eventually performs an action. Traces that satisfy this requirement are called **weakly fair**.

Another technicality that must be addressed is what it means for the environment to provide the same values to two different traces. Two traces have **consistent environment actions** as long as the first discrepancy (if any) is not due to an environment component. If the circuit were to output different values to the environment in two different traces, then subsequent actions of the environment might differ in these traces as well. In the FIFO example, two traces have consistent environment actions if the source component generates the same sequence of dual-rail values in both traces.

More formally, let c be a component. For each action of c, we can associate a tuple, (in_c, out_c) where in_c denotes the values of the inputs of c before the action and out_c denotes the values of the the outputs of c after the action. Let t and u be traces. Let t_c denote the sequence of tuples corresponding to actions of c in t, and likewise for u_c. Component c is consistent in t and u if t_c and u_c are identical, or if the first tuple that differs differs in the value of at least one input. Traces t and u have consistent environment actions if and only if every environment component is consistent in the two traces.

2.4 Traces of eligible designs

We can now state the theorems upon which our verification method is based. Informal sketches are provided instead of formal proofs. Detailed proofs are available in [Wei96]. The first theorem states that the order in which component actions are performed does not affect the validity of bounded local formulas for an eligible design.

Theorem 1 (Equivalence of traces) *Given an eligible design, let t and u be two weakly fair traces with the same initial state and consistent environment actions. Traces t and u are equivalent.*

The proof is based on the observation that if a trace reaches a state where two components, c and d are enabled, the same state is reached if the action of c is performed first and the action of d second, or if the actions are performed in the other order. Furthermore, traces that differ only by a swap of such adjacent actions can be shown to be equivalent.

Now, consider the first state where u performs a different action than t. Because u is weakly fair, the component that acted in t must eventually act in u. A finite sequence of swap operation will move this action to the same position in u as it was in t. Furthermore, the trace produced by these swaps is equivalent to u. In this manner, we can construct a trace that is equivalent to u and has an arbitrarily long prefix that is the same as t. Therefore, if t satisfies a bounded local formula B, then u satisfies B as well. $\square$

Recall that equivalence of traces was defined only for bounded local formulas. As in the FIFO example, most data-path circuits are specified using unbounded formulas. Below, we extend theorem 1 to a class of unbounded formulas that we call "periodic." This class is large enough to specify the behaviour of most practical designs.

Let B be a bounded local formula. Let C be the set of components that appear in B and let p be a function from C to the positive integers. The formula $\mathit{shift}(B, p, i)$ is obtained by replacing each local literal, v_c^k in B with the literal $v_c^{k+i*p(c)}$. The **periodic extension** of B by p is

$$\forall i \in \{0 \ldots +\infty\}.\mathit{shift}(B, p, i)$$

In the FIFO example, equation 2 is the periodic extension of the bounded local formula

$$\begin{array}{rrcl} & (x[4].t_{\mathrm{snk}}^1, x[4].f_{\mathrm{snk}}^1) & = & (x[1].t_{\mathrm{src}}^1, x[1].f_{\mathrm{src}}^1) \\ \wedge & (x[4].t_{\mathrm{snk}}^2, x[4].f_{\mathrm{snk}}^2) & = & (x[1].t_{\mathrm{src}}^2, x[1].f_{\mathrm{src}}^2) \end{array} \qquad (4)$$

by the period function $p(\mathit{src}) = 2$, $p(\mathit{snk}) = 2$.

Consider a design with initial state predicate G, and let C be a subset of the circuit and environment components. We say that the design has period p if and only if: (1) in any state satisfying G, only components in C are enabled; and (2) from every state in G and for every possible sequence of environment actions, there is a trace that returns to a state in G after each component c in C performs exactly $p(c)$ actions. Using these definitions, we can extend theorem 1 to the periodic extensions of bounded local formulas.

Continuing the FIFO example, the components appearing in equation 4 are *src* and *snk*. Because the initial state predicate in equation 3 allows any of *x[0].r*, … *x[4].r* to be false, components other than the source or sink can be enabled in the initial state. By strengthening the initial state predicate to

$$\bigwedge_{i=0}^{4} \neg x[i].t \wedge \neg x[i].f \wedge x[i].r \tag{5}$$

only the source component will be enabled in the initial state. Using this initial state predicate, the FIFO design is periodic with the period function given above.

Theorem 2 (Periodic local formulas) *Given an eligible design, let B be a bounded local formula of the design. Let C be the set of components that appear in B and let p be a period function for B. If the design has period p and all traces of the design satisfy B, then all traces of the design satisfy P, where*

$$P \quad = \quad \forall i \in \{0 \ldots + \infty\}.shift(B, p, i)$$

Proof sketch: For any initial state and sequence of environment inputs, there exists a trace that periodically satisfies the initial state predicate. It is straightforward to show that this trace satisfies P. By the equivalence of traces (theorem 1), all traces of the design must satisfy P. □

When using symbolic trajectory evaluation for verification, it is convenient to perform the actions of all enabled components as a single state transition. We now show that the validity and value of local formulas is preserved under such an execution model. Given a design, a **synchronous trace** is a sequence of states such that the first element of the sequence satisfies the initial state predicate of the design, and each pair of consecutive states in the sequence corresponds to performing the actions of *all* enabled components of the circuit and environment in a single trace. The final theorem presented here provides the basis for using synchronous traces to verify speed-independent designs.

Theorem 3 (Synchronous traces) *Given an eligible design, let L be a bounded or periodic local formula. L is satisfied for all weakly fair traces of the design if and only if it is satisfied for all synchronous traces.*

We first show that if L is satisfied for all synchronous traces, then L is satisfied for all traces. Let t be a weakly fair trace. By swapping actions, we can construct a trace u that is equivalent to u such that u is partitioned into epochs. The set of actions performed in each epoch is exactly the actions of those components that were enabled at the beginning of the epoch. Note that each component performs at most one action during each epoch. We construct a synchronous trace s by taking the first state from each epoch of u. It is straightforward to show that s is equivalent to u (and therefore equivalent to t). If all synchronous traces of the design satisfy L, then s must satisfy L, and therefore t satisfies L as well. The proof for the other direction is similar. $\square$

3 Implementation

Given a design, an initial state predicate, a bounded local formula, and a period function, the verification task is to show that: (1) the design is eligible; (2) for all allowed initial conditions and environment actions there is a trace which satisfies the bounded local formula; and (3) the design is periodic with the given period function. As shown in the section 2.4, verifying these three properties establishes that all traces of the design satisfy the periodic extension of the local formula. In our implementation, we use symbolic model checking to verify that the design is eligible and symbolic trajectory evaluation to verify the other two properties.

3.1 Model checking

Symbolic model checking [BCo+94] can be used to show that a property holds for all reachable states of a circuit. This approach is fully automatic, and is computationally feasible for relatively small designs. Each component is separately tested using an environment that observes the handshaking protocols but can apply arbitrary data values. In this context, the model checker verifies that the component is speed-independent, has stable inputs (if required), and has no output choice (if it is a circuit component). We also verify that the component complies with its side of the handshaking protocols. Because we are only concerned about reachable states (i.e. safety properties), verifying that each component is eligible in a generic context ensures that the entire circuit is eligible (This approach to compositional verification is described formally by Abadi and Lamport [AL89]). Because each component is verified separately, model checking is only applied to small circuits, and we avoid the problem of state space explosion. This shows that the circuit is eligible.

3.2 Symbolic trajectory evaluation

Symbolic trajectory evaluation (STE) [SB95] is a verification technique for circuits with a deterministic next state function and specifications written in a restricted logic called "trajectory formulas." STE does not compute the reachable subset of the state space and can be applied successfully to much larger circuits than model checking. Although restrictive, trajectory formulas readily express requirements of the form "if the initial state satisfies a given predicate then the sequence values output by the circuit are a given function of the sequence of values applied at the inputs," where the sequences are finite. This corresponds naturally to the requirements that we write as local formulas, allowing us to show that for any allowed initial state and environment action there is a trace satisfying the desired bounded local formula.

Verifying that a design has period function p is similar. If c is a component in C_p, the domain of p, we augment the model for c with counters that indicate how many operations it has performed. Furthermore, we modify the model of c to disable further operations after performing $p(c)$ actions. Let G be the initial state predicate of the circuit. If the original circuit had a trajectory that returns to G with the correct period, the modified circuit will stop in that state (no further state transitions will be possible). Using STE, we verify that all trajectories that start in a state that satisfies the initial state predicate, G, end in a state that also satisfies G, and that for each component c in C_p, the counter for c has value $p(c)$. This shows that the design is periodic, concluding the verification.

Both the model checker and STE implementation use Ordered Binary Decision Diagrams (OBDDs [Bry92]) to represent boolean formulas. Because we only have to model check small circuits, this approach was very effective. However, this is likely to be a bottleneck when verifying data paths larger than those considered here. We address this point more in sections 4.

3.3 Design representation

The designs are modeled at the component level using VHDL. Each component is described using "behavioural" VHDL, and these are combined to form the complete design using "structural" VHDL. The Voss system for STE [SB95] supports VHDL design descriptions. Modifying the models of environment components to test periodicity as described above is easily done in VHDL. For the model checking part of the verification, we use the st2fl model checker [LGS94]. Translation from VHDL to ST for the model checker is manual, but straightforward. Local formulas are written as trajectory formulas for the Voss system, and we extended the st2fl model checker to automatically verify the eligibility of designs.

4 Vector Multiplier

For a data path with significant complexity, we chose the vector multiplier described in [SS93]. This circuit computes the inner product of pairs of vectors using an array of "Multiply-Accumulate-Shift" blocks, with one block for each bit of the final result. The accumulating sum is stored in carry-save form until it is output. The computation of an inner product is a nested iteration: the outer loop performs one iteration for each element of the vectors, and the inner loop performs one iteration for each bit of one of the vectors. Initially, the sum and carry bits of the accumulator are set to zero and first element of one vector is right justified in the array. The corresponding element of the other vector is applied serially with the least significant bit first. For each bit of the serial operand, a partial product is computed and added to the accumulating sum. The parallel operand and carry word are shifted to the left, and the next bit of the serial operand is applied. After the entire serial operand has been applied, the next parallel operand is entered into the MAS blocks, again right justified, and another product is computed. When all of the products have been computed, the serial operand is set to zero for as many iterations of the inner loop as there are bits in the accumulator. This ensures that the carry word is zero and the result is ready to be output.

A diagram of the MAS bit-slice is shown in figure 3. Unlabeled devices in the diagram represent asynchronous latches (FIFO cells). There is a source (labeled with a 0) which produces a stream of dual-rail false values, and a sink (labeled with a ●). The **And** cell and **Adder** perform the partial product calculation and accumulation as described above, and the three rows of FIFO cells perform pipeline buffering. Cells **M1** and **M2** are multiplexors which control the nested iterations. These multiplexors are asymmetric: when the control line is a dual-rail false, data is passed directly through; when the control line is true, the lower input is passed to the upper output and no operation is performed on the other ports.

The two control signals, `Ctl1` and `Ctl2`, control the iterations. At the beginning of an inner product computation, both signals are true: **M1** loads a bit from the parallel operand, and **M2** sets the accumulated value to zero.

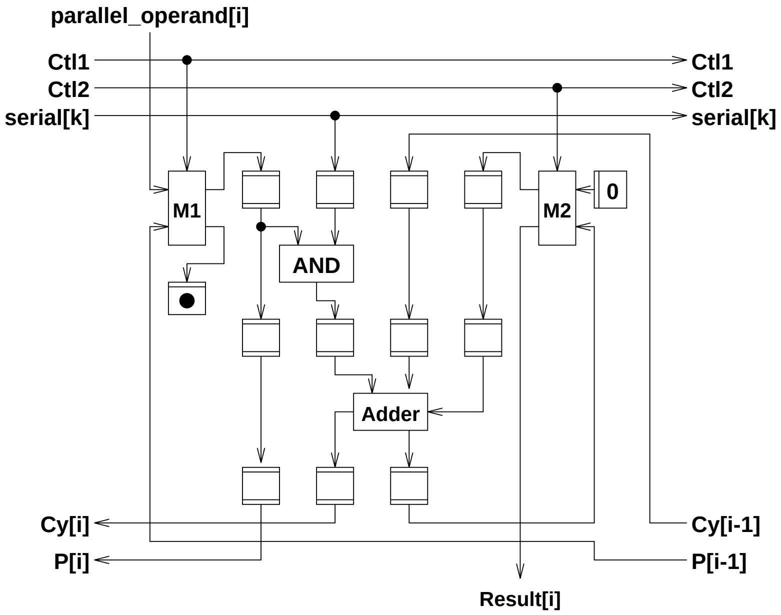

Figure 3: Bit Slice of the MAS Block (adapted from [SS93])

During each cycle of a multiplication, both control signals are false: **M1** receives the next less significant bit of the parallel operand, and **M2** returns the sum to the adder for another cycle of accumulation. When a multiplication is complete, `Ctl1` is set to true and the `M1` cells load a new parallel operand into the MAS array, and `Ctl2` is false, allowing the current accumulation to continue. When the entire inner product has been computed, both control signals are set to true again, the result is output, the accumulator is reset to zero, and a new parallel operand is loaded.

This nested iteration complicates the formalism developed in the preceding sections. Theorem 2 requires a period function p in order to show that we can verify unbounded traces. For this design, there is no clear p. The accumulator is only returned to its initial state (zero) after the complete inner product is computed. However, the circuit accommodates vectors of arbitrary length, and the number of operations performed inner product calculation depends on the length of the vectors. Thus, we chose to verify the design for various fixed length vectors. Table 1 below shows verification times for short vectors. The rapid increase in verification times with the number of pairs is a result of the OBDD representations of the accumulated sum. An

OBDD's size increases considerably when its value depends on the interaction of many variables, and the most significant bits of the accumulated sum depend in a complicated fashion on all of the bits of each operand pair.

Device	STE Time (in s)
FIFO (7 stages)	3
Adder (4 bit)	4
Vector Multiplier (1 pair)	14
Vector Multiplier (2 pairs)	22
Vector Multiplier (3 pairs)	602
Vector Multiplier (4 pairs)	stopped after 340 min

Table 1: STE Verification Times

Initially, we had hoped to define the periodicity according to the computation of a single scalar multiply-accumulate. A difficulty arises due to the fact that the MAS block may or may not output the result at the end of this computation according to the value off `Ctl2`. To verify this interaction with the environment, we must include an environment component that receives the result in the local formula, but this formula may perform two or zero operation depending on whether or not this multiplication was the last computation of the inner product. Accordingly, there is no suitable period function for the inner iteration.

5 Conclusions

The sequence of operations performed by each component of a speed-independent design without output choice is independent of the relative timing of the various components in the design. We exploited this observation by introducing local formulas. Using these formulas, data-path circuits are naturally specified according to the relationships that must hold between the sequences of values input and output by the circuit. This leads to an automatic verification approach that can be applied to designs of significant complexity. In our implementation, speed-independence is verified by model checking each component separately, and the local formulas are verified by applying symbolic trajectory evaluation to an equivalent, synchronous model of the circuit behaviour.

Many asynchronous designs from the literature satisfy the speed-independence conditions used in this paper. In particular, we have demonstrated the practicality of our approach by verifying a vector multiplier chip from the literature. We are exploring ways to allow period functions that depend on data values to overcome the restrictions described in section 4. We are also interested in applying these techniques to circuits with timing dependencies. If the circuit is hazard-free, then it should have the same properties with respect to local formulas as a speed-independent circuit. However, the simple model checking techniques that we used to show that circuits were speed-independent cannot be used to show that a circuit is hazard free.

Acknowledgements

The authors are grateful to Carl Seger for providing the Voss system and providing many helpful suggestions for implementing the techniques described in this paper. We also appreciate assistance from Jens Sparsø who provided us with the vector multiplier design and explained many details of its operation. This work was supported by NSERC Research Grant 138501, a fellowship from the B.C. Advanced Systems Institute, and equipment grants from Digital Equipment Corporation and Sun Microsystems.

References

[AL89] Martín Abadi and Leslie Lamport. Composing specifications. In J.W. de Bakker et al., editors, *Proceedings of the REX Workshop, "Stepwise Refinement of Distributed Systems"*. Springer-Verlag, 1989. LNCS 430.

[AS95] Mark D. Aagaard and Carl-Johan Seger. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In *Proceedings of the 1995 ICCAD*, November 1995.

[BCo+94] J.R. Burch, E.M. Clarke, David E. Long others, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. 13(4):401–424, April 1994.

[Bea93] Derek L. Beatty. *A methodology for formal hardware verification, with application to microprocessors*. PhD thesis, Computer Science Deparatment, Carnegie Mellon University, 1993.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[Dar94] Mohammad Darwish. Formal verification of a 32-bit pipelined RISC processor. Master's thesis, Department of Electrical Engineering, University of British Columbia, 1994.

[LGS94] Trevor W.S. Lee, Mark R. Greenstreet, and Carl-Johan Seger. Automatic verification of asynchronous circuits. *IEEE Design and Test*, 12(1):24–31, Spring 1994.

[MB59] David E. Muller and W.S. Bartky. A theory of asynchronous circuits. In *Proc. Int. Symp. Theory Switching*, pages 204–243, 1959.

[MB+89] Alain J. Martin, Steven M. Burns, et al. The design of an asynchronous microprocessor. In *Proceedings of the Conference on Advanced Research in VLSI*, Caltech, 1989.

[SB95] Carl-Johan Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2), March 1995.

[Seg93] Carl-Johan Seger. Voss — A formal hardware verification system: User's guide. Technical Report UBC-CS-93-45, Department of Computer Science, University of British Columbia, Vancouver, BC, 1993.

[SS93] Jens Sparsø and Jørgen Staunstrup. Delay-insensitive multi-ring structures. *INTEGRATION*, 15(3):313–340, October 1993.

[SSM94] Robert F. Sproull, Ivan E. Sutherland, and Charles E. Molnar. The counterflow pipeline processor architecture. *IEEE Design and Test*, 11(3):48–59, Fall 1994.

[Wei96] David T. Weih. Formal verification of asynchronous data-path circuits. Master's thesis, Department of Computer Science, University of British Columbia, April 1996.

[Wil91] Ted E. Williams. *Self-Timed Rings and their Application to Division.* PhD thesis, Stanford University, May 1991.