Verifying Nondeterministic Implementations of Deterministic Systems[1]

Alok Jain
Department of ECE
Carnegie Mellon University
Pittsburgh, PA 15213
email: alok.jain@ece.cmu.edu

Kyle Nelson
IBM Corporation
AS/400 Division
Rochester, MN 55901
email:kln@vnet.ibm.com

Randal E. Bryant
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
email:randy.brant@cs.cmu.edu

Abstract. Some modern systems with a simple deterministic high-level specification have implementations that exhibit highly nondeterministic behavior. Such systems maintain a simple operation semantics at the high-level. However their underlying implementations exploit parallelism to enhance performance leading to interaction among operations and contention for resources. The deviation from the sequential execution model not only leads to nondeterminism in the implementation but creates the potential for serious design errors. This paper presents a methodology for formal verification of such systems. An abstract specification describes the high-level behavior as a set of operations. A mapping relates the sequential semantics of these operations to the underlying nondeterminism in the implementation. Symbolic Trajectory Evaluation, a modified form of symbolic simulation, is used to perform the actual verification. The methodology is currently being used to verify portions of a superscalar processor which implements the PowerPC architecture. Our initial work on the fixed point unit indicates that this is a promising approach for verification of processors.

1. Introduction

Some modern circuit designs with a simple deterministic high-level specification have implementations that exhibit highly nondeterministic behaviors. Systems that operate on an externally visible stored state such as memories, data paths and processors often exhibit these behaviors. The implementation of these systems frequently overlap the execution of tasks in an effort to enhance performance while maintaining the appearance of sequential execution.

A large class of systems that exhibit such behavior are processors. At the high-level, the sequencing model inherent in processors is the sequential execution model. However, the underlying implementation of processors use pipelines, multiple instruction issue, and nondeterministic protocols to interact with other subsystems. The resulting interaction among instructions and contention for resources leads to nondeterminism in the implementation. Such implementations contain many subtle features with the potential for serious design errors. The methodology outlined in this paper is able to bridge the wide gap between the abstract specification and the implementation's often radical deviation from the sequential execution model.

A methodology for formal verification must ensure that such a system functions correctly under all possible execution sequences. Since there is an infinite number of execution sequences, we verify each operation individually and then reason about stitching arbitrary operations together to form execution sequences.

The goal is to develop a methodology with which a designer can show that an imple-

1. This work partially funded by Semiconductor Research Corporation # 95-DC-068.

mentation correctly fulfills an abstract specification of the desired system behavior. The abstract specification describes the high-level behavior of the system independent of any timing or implementation details. As an example, the natural specification of a processor is the instruction set architecture. The specification is a set of *abstract assertions* defining the effect of each operation on the user-visible state. The verification process must bridge a wide gap between the detailed circuit and the abstract specification. In spanning this gap, the verifier must account for issues such as system clocking, pipelines and interfaces with other subsystems. To bridge this gap between the abstract behavior and the circuit, the verification process requires some additional mapping information. The mapping defines how state visible at the abstract level is realized in the detailed circuit. The mapping has both spatial and temporal information. Designers are typically aware of the mapping but do not have a rigorous way of recording or documenting the information.

Our specification is thus divided into two components: the *abstract specification* and the *mapping information*. The distinction serves several purposes. Several feasible circuit designs can be verified against a single abstract specification. An abstract specification can be used to verify both an unpipelined and a pipelined version of a processor. The abstract specification describes the instruction set of the processor independent of any pipeline details. The task of the mapping is to relate the abstract specification to the complex temporal behavior and nondeterministic interactions of the pipelined processor. As an example, an instruction might stall in a pipeline stage waiting to obtain the necessary resources. The order and timing in which these resources are obtained often varies leading to nondeterministic behavior. Our verification methodology will verify the circuit under all possible orders and timing.

The distinction between the abstract specification and mapping enables hierarchical verification. This paper concentrates on a single level of mapping that maps the abstract specification into a specific implementation. In the future, one can envision an entire series of implementation mappings. Each level in the mapping serves to make the assertion more concrete. A verification task could be performed at each level. A series of mappings could also be used to perform modular simulation. Simulation models could be developed at each abstraction level and models at different levels of abstractions could be intermixed using the mapping information.

Once the abstract assertions have been individually verified, the methodology must be able to stitch operations together in order to reason about execution sequences. The mapping has information about how to stitch instructions together. Since the abstract assertions use a sequential execution model, stitching operations at the abstract specification level requires a simple sequencing of the instructions. However, the underlying nondeterministic implementation requires operations to interact and overlap in time. The mapping ensures that the operations can interact and overlap to create arbitrary execution sequences.

The abstract specification and the mapping are used to generate the *trajectory specification*. The trajectory specification consists of a set of *trajectory assertions*. Each abstract assertion gets mapped into a trajectory assertion. The verification task is to verify the set of trajectory assertions on the circuit. A modified form of symbolic sim-

ulation called *Symbolic Trajectory Evaluation*[1] is used to perform the verification task. We use the term trajectory specification and trajectory assertions partly for historical reasons. Our trajectory assertions are a generalization of the trajectory assertions introduced by Seger[4]. The justification is that the assertions define a set of trajectories in the simulator.

The formal verification methodology presented in this paper is currently being used to verify a superscalar processor which implements the PowerPC architecture[12]. The processor has several complex features such as pipeline interlocks, multiple instruction issue, and branch prediction to advance the state of the art in verification.

1.1. Related Work

Beatty[2][3] laid down the foundation for our methodology for formal verification of processors. The instruction set was specified as a set of declarative abstract assertions. A downward implementation mapping was used to map discrete transitions in the abstract specification into overlapping intervals for the circuit. The overlapping was specified by a *nextmarker* which defined the nominal end of the current instruction and the start of the next instruction. However this work had one basic limitation. The verification methodology could handle only bounded single behavior sequences. The mapping language was formulated in terms of a formalism called *marked strings*[2]. Marked strings could not express divergent or unbounded behavior.

We have extended the verification methodology so as to handle a greater level of nondeterministic behavior. As a motivating example, consider a fixed point unit of a processor performing a bitwise-or operation that fetches two source operands (`A` and `B`) from a dual-ported memory subsystem. Assume that the operands might not be immediately available. The fixed point unit might have to wait for an arbitrary number of cycles before either of the operands are available. Furthermore the operands might be received in different orders: Operand `A` might arrive before `B`, operand `B` might arrive before `A`, or both might arrive simultaneously. The verification methodology should be able to verify the correctness of the circuit under any number of wait cycles and all possible arrival orders. Marked strings cannot express such an operation. Our formulation is in terms of state diagrams. State diagrams allow users to define unbounded and divergent behavior.

Symbolic Trajectory Evaluation (STE) has been used earlier to verify trajectory assertions. Beatty[3] mapped each abstract assertion into a set of symbolic patterns. STE was used to verify the set of symbolic patterns on the circuit. The set of symbolic patterns corresponded to a single sequence of states in a state diagram. Seger[4] extended STE to perform fixed point computations to verify a single sequence of states augmented with a limited set of loops. In our work, trajectory assertions are general state diagrams. We have extended STE to deal with generalized trajectory assertions.

Our work has some resemblance to the capabilities provided by the Symbolic Model Verifier (SMV)[5][6]. SMV requires a closed system. The environment is modeled as a set of machines. In some sense the state diagrams in our mapping correspond to creating an environment around the system. The state diagrams corresponding to the inputs can be viewed as generators that generate low-level signals required for the operation

of the processor. State diagrams corresponding to outputs can be viewed as acceptors that recognize low-level signals on the outputs of the processor. However, there is one essential difference. Though SMV does provide the capability of describing the environment, it does not provide a methodology for rigorously defining these machines and stitching them together to reason about infinite execution sequences. The other difference is that the model-checking algorithm in SMV requires the complete next-state relation. It would be impossible to obtain the entire next-state relation for a complex processor. On the other hand, we use STE to evaluate the next-state function on-the-fly and only for that part of the processor that is required by the specification.

Kurshan[7][8] has used the concept of mappings to perform hierarchical verification. The specification (called a *task*) is represented by a deterministic automaton. The implementation is modeled as a state machine. Verification is cast in terms of testing whether the formal language of the implementation is contained in the formal language of the task. The user can provide reduction transformations as a basis of complexity management and hierarchical verification. Reductions are homomorphic transformations which correspond to abstraction mappings. Inverse reductions (called *refinements*) correspond to our implementation mappings. However, Kurshan does not have the concept of stitching tasks together to verify an infinite sequence of tasks. Also, Kurshan uses language containment as opposed to Symbolic Trajectory Evaluation as the verification task.

Burch[13] has proposed a verification methodology that avoids the need to explicitly define the mapping. Instead, an abstraction mapping is implicitly defined by flushing the pipeline and then using a projection function to hide some of the internal state. The methodology has been used to verify a pipelined version of the DLX architecture. However, the abstraction mappings are limited to internal states in the processor. In addition to mapping internal states, our implementation mappings allow users to define nondeterministic input/output protocols.

In the area of processor verification theorem proving has been used in the past to verify processors[9][11]. However, in these cases either the processor was very simple or a large amount of manual intervention was required. STE was used to verify the Hector processor[3]. However, Hector is a very simple processor similar to the PDP-11[10]. We are using the verification methodology to verify the PowerPC architecture. Recently, we verified the pipeline interlocks in the decode stage of the fixed point unit.

Details of how to specify the system behavior in our methodology are discussed in Section 2. The system behavior is specified in terms of a set of abstract assertions and the implementation mapping. An abstract assertion and the corresponding mapping for a small example are shown in Section 3. The abstract assertions are mapped into a set of trajectory assertions. Details of how to verify the set of trajectory assertions on a circuit are discussed in Section 4. Some of the ongoing work on using the formal verification methodology to verify a superscalar processor is presented in Section 5.

2. Specification

The specification consists of the abstract specification and the mapping. The mapping is a nondeterministic mapping formulated in terms of a variation of state diagrams

called *control graphs*.

2.1. Abstract Specification

Hardware Description Languages seem to be the obvious choice for expressing abstract specifications. However, Hardware Description Languages tend to overspecify systems since they tend to describe a particular implementation rather than an abstract specification. It seems that these languages are appropriately named. They “describe” rather than “specify” hardware. We have defined a *Hardware Specification Language* to specify hardware. Here we are presenting the basic form of the Hardware Specification Language. We will later in this section extend the language to the symbolic and vector domains. The specification is an abstract machine. The abstract machine is associated with a set of single bit *abstract state elements* (S_v). Transitions in the machine are described as a set of *abstract assertions*. Each abstract assertion is an implication of the form: $P \Rightarrow Q$, where P is the precondition and Q is the postcondition. P and Q are abstract formulas of the form:

- **Simple abstract formula**: (s_i **is** 0) and (s_i **is** 1) are abstract formulas if $s_i \in S_v$.
- **Conjunction**: (F_1 **and** F_2) is an abstract formula if F_1 and F_2 are abstract formulas.
- **Domain restriction**: $(0 \rightarrow F)$ and $(1 \rightarrow F)$ are abstract formulas if F is an abstract formula.

Mathematically an abstract assertion can be defined as follows: Assume S is the set of assignments to the abstract state elements. Therefore $S = \{0, 1\}^n$, where $n = |S_v|$. Define a satisfying set for an abstract formula as the set of abstract state assignments that satisfy the abstract formula. The satisfying set of an abstract formula F, written $\mathcal{Sat}(F)$, is defined recursively:

- $\mathcal{Sat}(s_i\ \textbf{is}\ 0)$ is a subset of S with 2^{n-1} abstract state assignments with the ith position in each assignment being 0. Similarly $\mathcal{Sat}(s_i\ \textbf{is}\ 1)$ is a subset of S with 2^{n-1} abstract state assignments with the ith position in each assignment being 1.
- $\mathcal{Sat}(F_1\ \textbf{and}\ F_2) = \mathcal{Sat}(F_1) \cap \mathcal{Sat}(F_2)$.
- $\mathcal{Sat}(0 \rightarrow F) = S$, and $\mathcal{Sat}(1 \rightarrow F) = \mathcal{Sat}(F)$.

Each assertion defines a set of transitions in an abstract Moore machine. For an abstract assertion $A = P \Rightarrow Q$, the sets $\mathcal{Sat}(P)$ and $\mathcal{Sat}(Q)$ are the sets of abstract state assignments that satisfy the precondition and postcondition respectively. The transitions associated with the assertion can now be defined as: $\mathcal{Trans}(A) = (\mathcal{Sat}(P) \times \mathcal{Sat}(Q)) \cup ((S - \mathcal{Sat}(P)) \times S)$. If the abstract machine starts in a set of states that satisfy the precondition, then the machine should transition into a set of states that satisfy the postcondition. The set $\mathcal{Sat}(P) \times \mathcal{Sat}(Q)$ represents the set of transitions when the precondition is satisfied. If the machine starts in a set of states that do not satisfy the precondition, then the assertion does not place any restrictions on the set of transitions. The set $(S - \mathcal{Sat}(P)) \times S$ represents the set of transitions when the precondition is not satisfied.

Let i index the set of abstract assertions. The intersection $\bigcap_i \mathcal{Trans}(A_i)$ defines a nondeterministic Moore machine corresponding to the abstract specification.

Consider the example of a bitwise-complement operation. A bitwise-complement operation requires two abstract state elements, `SA` and `ST`, where `SA` is the source operand and `ST` is the target operand. Assuming `SA` and `ST` are single bit operands, the bitwise-complement operation can be specified by using 2 abstract assertions.

$(\mathtt{SA}\ \mathbf{is}\ 0) \Rightarrow (\mathtt{ST}\ \mathbf{is}\ 1)$ // Assertion A_1

$(\mathtt{SA}\ \mathbf{is}\ 1) \Rightarrow (\mathtt{ST}\ \mathbf{is}\ 0)$ // Assertion A_2

For this example, $n = 2$ and the set S has 4 state assignments, $S = \{00, 01, 10, 11\}$, where the left bit is the logic value associated with `SA` and the right bit is the logic value associated with `ST`. The transitions corresponding to these assertions are shown in Figure 1. Let us consider assertion A_1. Now $\mathcal{S}at(\mathtt{SA}\ \mathbf{is}\ 0) = \{00, 01\}$ and $\mathcal{S}at(\mathtt{ST}\ \mathbf{is}\ 1) = \{01, 11\}$. The solid edges represent transitions when the precondition is satisfied, i.e. the set $\mathcal{S}at(P) \times \mathcal{S}at(Q)$. The shaded edges represent the transitions when the precondition is not satisfied i.e. the set $(S - \mathcal{S}at(P)) \times S$. A similar case can be made for assertion A_2. The intersection of the transitions associated with both assertions gives the Moore model for the specification. In this particular model, nondeterminism is used only to represent the choice of the next input. In other cases, nondeterminism can be used to represent unspecified or don't care behavior such as the output of the multiplier while performing an add operation in a processor.

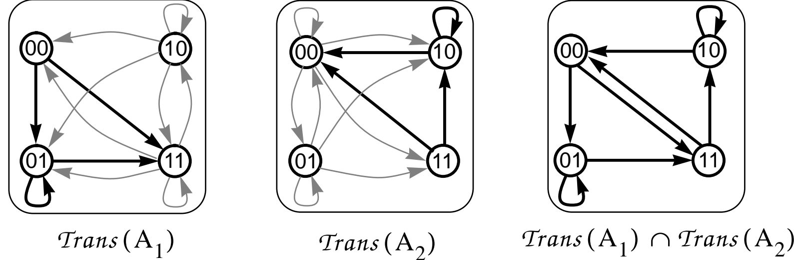

Figure 1. Nondeterministic Moore machine model for assertions.

We have presented the basic form of the abstract assertions. They can be extended to the symbolic and vector domain. In the symbolic domain, the domain restriction in abstract formulas is extended to be of the form: $(e \rightarrow F)$ where e is a Boolean expression over a set of symbolic variables. We introduce the notation $(s_i\ \mathbf{is}\ e)$ as a shorthand for $(\bar{e} \rightarrow (s_i\ \mathbf{is}\ 0))$ **and** $(e \rightarrow (s_i\ \mathbf{is}\ 1))$. In the vector domain, the abstract state elements are extended to represent different word sizes. A symbolic abstract assertion defines a set of scalar abstract assertions. A bitwise-complement operation for k-bit word size can be specified with a single symbolic abstract assertion. Assume that `a` is a symbolic k-bit variable that denotes the current value of the source operand. The symbolic abstract assertion for the bitwise-complement operation is:

$(\mathtt{SA}\ \mathbf{is}\ \mathtt{a}) \Rightarrow (\mathtt{ST}\ \mathbf{is}\ \tilde{\mathtt{a}})$

The ~ operator is the bitwise-complement operator. The symbolic abstract assertion represents 2^k scalar assertions corresponding to all possible assignments to the variable `a`.

2.2. Control Graphs

Control graphs are state diagrams with the capability of synchronization at specific time points. Mathematically a control graph can be represented as a tuple: $G = \langle V, U, E, s, t\rangle$, where

- V is the set of *state vertices*.
- U is the set of *event vertices*.
- E is the set of directed edges.
- s is the source, $s \in U$.
- t is the sink, $t \in U$.

There are two types of vertices in a control graph: 1) Event vertices representing instantaneous time points and 2) State vertices representing some non-zero duration of time. Event vertices are used for synchronization. A control graph has a unique event vertex with no incoming edges called the source vertex. Also the graph has a unique event vertex with no outgoing edges called the sink vertex. Nondeterminism is modelled as multiple outgoing edges from a vertex.

2.3. Implementation Mapping

The implementation mapping provides a spatial and temporal mapping for the abstract machine. The mapping has three main components:

- Set of *node elements*.
- Single *main machine.*
- Set of *map machines.*

The mapping defines a set of node elements (N_v) that represent inputs, outputs and internal states in the circuit.

The main machine defines the flow of control for individual system operation. However we need to reason about sequence of operations. Therefore, the main machine also defines how individual operations can be stitched together to create valid execution sequences. The main machine is a tuple of the form: $M = \langle V_m, U_m, E_m, s_m, t_m, Y_m\rangle$, where

- $\langle V_m, U_m, E_m, s_m, t_m\rangle$ is a control graph.
- Y_m is a vertex cut of the control graph, $Y_m \subseteq U_m$.

The source s_m denotes the start of an operation and the sink t_m denotes the end of the operation. The vertex cut Y_m denotes the *nominal end* of the operation. The nominal end of an operation is defined to be the timepoint where the next operation can be started. The vertex cut divides the set of vertices into 3 sets, X_m, Y_m and Z_m where $X_m \cup Y_m \cup Z_m = V_m \cup U_m$. The set X_m represents the set of vertices to the left of the vertex cut and the set Z_m represents the set of vertices to the right of the vertex cut. The vertex cut places certain restrictions on the set of edges in the graph. All paths from vertices in X_m to vertices in Z_m have to pass through an event vertex in Y_m. In addition there cannot be any paths from vertices in Z_m to vertices in X_m. The restriction ensures that the pipeline is committed to an instruction once it has started.

For an unpipelined circuit, $Y_m = \{t_m\}$, since the current operation must end before

the next operation can start. However, for pipelined circuits, Y_m is a cutset of event vertices between the source and sink. The cutset defines the time point when the next operation can be started, thus overlapping the current and next operations. As an example, consider the main machine for a simple pipelined processor shown in Figure 2. The vertices F, D, and E are state vertices representing the fetch, decode and execute stages respectively. The vertices s, p, q, and t are event vertices. Note that the nominal end cutset indicates that the next operation can be fetched when the current operation has entered the decode stage.

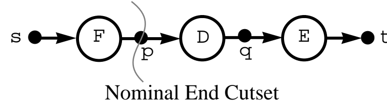

Figure 2. Main machine for a simple pipelined processor.

Individual instructions can be stitched together to create execution sequences as shown in Figure 3. The machines M^1, M^2, M^3 are multiple copies of the main machine with the superscripts indexing successive instructions. The machine M^* corresponds to the infinite execution sequence obtained from stitching all instructions. Stitching is performed by aligning the source vertex of an instance of the main machine with the nominal end cutset of the previous main machine and taking the cross-product of all the aligned machines.

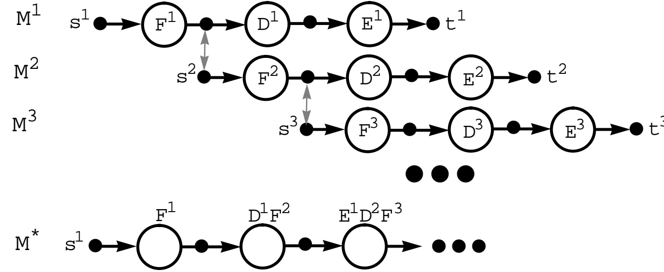

Figure 3. Stitching main machines together to form execution sequences.

In addition to the main machine, each simple abstract formula in the abstract specification is mapped into a map machine. The map machine provides a spatial and temporal mapping for the simple abstract formula. A control graph defines the temporal component of the mapping. Each state vertex in the control graph is associated with a node formula. The node formulas define the spatial component of the mapping. Here we will present the basic form of the node formulas. Later in the section we will extend them to the symbolic and vector domain. Node formulas are of the form:

- **Simple node formula**: (n_i **is** 0) and (n_i **is** 1) are node formulas if $n_i \in N_v$.
- **Conjunction**: (F_1 **and** F_2) is a node formula if F_1 and F_2 are node formulas.
- **Domain restriction**: ($0 \rightarrow F$) and ($1 \rightarrow F$) are node formulas if F is a node formula.

The map machines are not completely independent. A synchronization function maps event vertices in the map control graph to event vertices in the main control graph. The synchronization function relates a simple abstract formula with the flow of control of

the entire operation. Mathematically a map machine for a simple abstract formula s can be represented as a tuple of the form: $\mathcal{M}ap(s) = \langle V_s, U_s, E_s, s_s, t_s, \sigma_s, \Upsilon_s \rangle$, where

- $\langle V_s, U_s, E_s, s_s, t_s \rangle$ is a control graph.
- σ_s is a labelling function that labels state vertices with node formulas.
- Υ_s is the synchronization function, $\Upsilon_s : U_s \rightarrow U_m$.

We have presented the basic form of the node formulas and map machines. The node formulas can be extended to the symbolic and vector domain. In the symbolic domain, the domain restriction is extended to be of the form: $(e \rightarrow F)$ where e is a Boolean expression over a set of symbolic variables. We introduce the notation $(n_i \textbf{ is } e)$ as a shorthand for $(\bar{e} \rightarrow (n_i \textbf{ is } 0))$ **and** $(e \rightarrow (n_i \textbf{ is } 1))$. In the vector domain, node elements are extended to represent a vector of nets in the circuit. Similarly, map machines can be extended to the symbolic and vector domains. A symbolic map machine defines a set of scalar map machines. The mapping for a k-bit operand (ST) can be specified by a single symbolic machine, $\mathcal{M}ap(\mathtt{ST} \textbf{ is } \mathtt{v})$, where v is a k-bit symbolic variable. The symbolic machine captures 2^k scalar map machines corresponding to all possible assignments to the variable v.

The trajectory formula corresponding to an abstract formula is the set of trajectories defined by the abstract formula for a particular implementation. The user needs to define the map machine for simple abstract formulas. Given the map machines, the trajectory formula for an abstract formula AF, written $\mathcal{T}raj(AF)$, can be automatically computed as follows:

- $\mathcal{T}raj(s_i \textbf{ is } 0) = \mathcal{M}ap(s_i \textbf{ is } 0)$ and $\mathcal{T}raj(s_i \textbf{ is } 1) = \mathcal{M}ap(s_i \textbf{ is } 1)$.
- $\mathcal{T}raj(AF_1 \textbf{ and } AF_2) = \mathcal{T}raj(AF_1) \parallel \mathcal{T}raj(AF_2)$, where $\parallel$ is the parallel composition operator. Parallel composition amounts to taking the cross product of the two control graphs under restrictions specified by the synchronization function. Assume that v_1 and v_2 are state vertices in the trajectory formulas for AF_1 and AF_2 respectively. Further assume that NF_1 and NF_2 are the node formulas associated with vertices v_1 and v_2 respectively. Then the node formula associated with the cross-produce vertex $(v_1 \times v_2)$ is $(NF_1 \textbf{ and } NF_2)$.
- $\mathcal{T}raj(e \rightarrow AF)$ is essentially the same as $\mathcal{T}raj(AF)$, except that the node formulas are modified. Assume that v is a vertex in the trajectory formula for AF with node formula NF. The node formula associated with the vertex v for the trajectory formula $(e \rightarrow AF)$ is $(e \rightarrow NF)$.

Note that we can verify each operation individual and then reason about stitching operations together since $\mathcal{T}raj(AF)$ captures the effect of any preceding or following operations.

For an abstract assertion $A = P \Rightarrow Q$, $\mathcal{T}raj(P)$ and $\mathcal{T}raj(Q)$ are the trajectory formulas associated with the precondition and postcondition respectively. The trajectory assertion corresponding to A can be automatically computed as $\mathcal{T}raj(P) \mathbin{/\!/} \mathcal{T}raj(Q)$, where $/\!/$ is the shift-and-compose operator. The shift-and-compose operator shifts the start of the machine $\mathcal{T}raj(Q)$ to the nominal end cutset in the main machine and then performs the composition. The node formulas in $\mathcal{T}raj(P)$ are treated as *antecedent*

node formulas. The antecedent node formulas define the stimuli and current state for the circuit. The trajectory formula corresponding to the precondition is being used as a generator since it generates low-level signals for the circuit. On the other hand, node formulas in $\mathcal{T}raj(Q)$ are treated as *consequent node formulas*. The consequent node formulas define the desired response and state transitions. The trajectory formula corresponding to the postcondition is being used as an acceptor since it defines the set of acceptable responses for the circuit.

Incidentally the shift and compose operator was also used to stitch operations together to form execution sequences in Figure 3, with the slight variation that the main machines were not associated with any node formulas.

3. Example

Assume that we want to verify the bitwise-or operation in an ALU. The abstract specification would define abstract state elements, `SA`, `SB` and `ST`. `SA` and `SB` serve as the source operands and `ST` serves as the target operand for the bitwise-or operation. Assume that `a` and `b` are symbolic variables that denote the current values of `SA` and `SB`. The abstract assertion for the bitwise-or operation is:

(`SA` **is** `a`) **and** (`SB` **is** `b`) ➡ (`ST` **is** `a|b`)

The | operator is the bitwise-or operator. Note that for a k-bit word size, this symbolic abstract assertion represents 4^k scalar assertions corresponding to all possible combinations of assignments to variables `a` and `b`.

The specification is kept abstract. It does not specify any timing or implementation specific details. Now let's assume a specific implementation of the ALU where the source operands have to be fetched from a register file and may not be immediately available. Assume that both source operands are associated with a valid signal which specifies when the operand is available. The valid signals have a default value of logic 0 and are asserted by the register file subsystem to a logic 1 when the operand is available. The ALU computes the bitwise-or and makes it available for storage back into the register file one cycle after both operands have been received. The block diagram for such an ALU is shown in Figure 4. This is a simple example that serves to illustrate some of the issues that we have encountered in our effort to verify the PowerPC architecture.

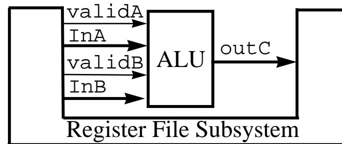

Figure 4. An implementation with a valid signal for each operand.

There are a few interesting points to note about this implementation. First, the ALU might have to wait for an arbitrary number of cycles before either of the source operands is available. Secondly, the source operands might arrive in different orders. Figure 5 shows part of an execution sequence for the implementation. The sequence is divided into various segments each of which represents a bitwise-or operation. A segment is associated with a nominal end marker. The execution sequence can be obtained

by overlapping the start of the successive segment with the nominal end marker of the current segment. Note that segments can be of different widths. Segment 1 exhibits a case where the A operand arrived before the B operand. Segments 2 and 3 exhibit cases where both operands arrived simultaneously. Segment 3 represents the maximally pipelined case where the operands were immediately available. And segment 4 exhibits a case where the B operand arrived before the A operand. The goal of verification is to ensure that the circuit correctly performs a bitwise-or operation under any number of wait cycles and under all possible arrival orders.

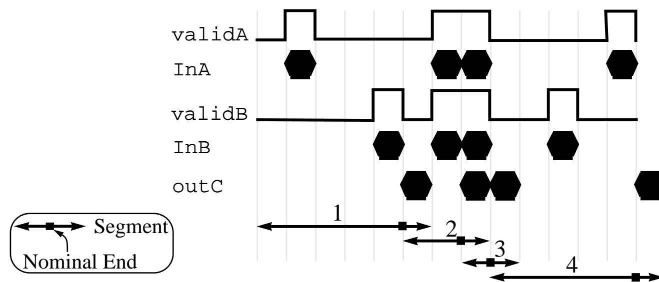

Figure 5. Part of an execution sequence for the bitwise-or operation.

The main machine for such an implementation is shown in Figure 6. Each state vertex represents a clock cycle. One or more cycles might be required to fetch the operands. This is represented by the state vertex fetch. Multiple cycles are required when the operands are not immediately available. After obtaining the operands, the result of the bitwise-or operation is available in the next cycle represented by state vertex execute. The control graph is modeling nondeterministic behavior. The machine can remain in the fetch state for an arbitrary number of cycles and then transition into the execute state. The main machine captures all possible segments in the execution sequence. The event vertex fetched represents the nominal end of the segment.

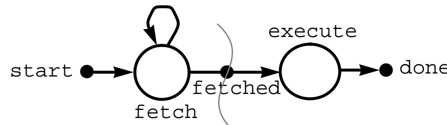

Figure 6. Main machine for the bitwise-or operation.

The map machine for the abstract formula (SA **is** v) is a nondeterministic machine shown in Figure 7. The symbolic variable v is serving as a formal argument. Later on the formal arguments will get replaced by the actual arguments. The labelling inside the state vertices represents the value of the valid signal for the A operand. The actual node formulas associated with the state vertices are shown in shadowed boxes in the figure. Since the operand might not be immediately available, the implementation might have to wait for an arbitrary number of cycles. This is represented by state vertex wait. The operand is received in state vertex fetch. After obtaining the A operand, the implementation might have to wait again for an arbitrary number of cycles for the B operand. This is represented by state vertex fill. The vertices M.start and M.fetched represent the event vertices start and fetched in the main machine. So the start of the map machine is synchronized to the start of the main machine. The

end of this machine is synchronized to the event vertex fetched in the main machine where M.fetched represents the time point where both operands have been fetched.

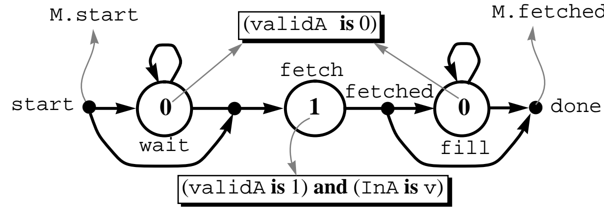

Figure 7. Map Machine for (SA is v).

The map machine for the abstract formula (SB **is** v) is the same as for state element SA except that the node formulas refer to the B operand.

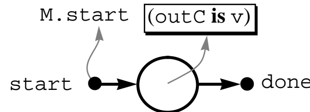

Figure 8. Map machine for (ST is v).

Finally the map machine for (ST **is** v) is shown in Figure 8. The start of this machine is synchronized with the start of the main machine. This indicates that the result of the computation for the previous operation should be available at the start of the current operation.

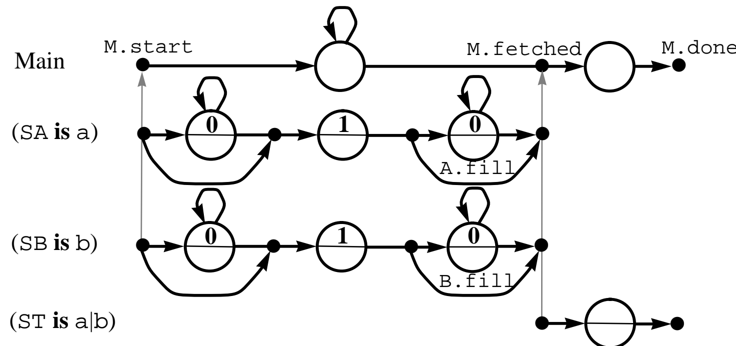

Figure 9. Alignment of machines for the bitwise-or operation.

The map machines get aligned with the main machine as shown in Figure 9. The abstract formulas (SA **is** a) and (SB **is** b) appear in the precondition of the abstract assertion. The formal argument in the corresponding map machine gets replaced by the actual arguments a and b. The node formulas in the map machines are treated as antecedent node formulas. Antecedent node formulas are shown in the upper half of the state vertex. The event vertices are synchronized as specified by the synchronization function. The abstract formula (ST **is** a|b) appears in the postcondition. The formal argument v is replaced by the expression a|b. The node formulas are treated as consequent node formulas. Consequent node formulas are shown in the lower half of the state vertex. The synchronization point is shifted to the nominal end cutset in the main machine.

The trajectory assertion for the bitwise-or operation corresponds to the composition of these machines. For this example, composition amounts to performing the cross product construction of the main, `SA` and `SB` machines between event vertices `M.start` and `M.fetched`, followed by the cross-product construction of the main and `ST` machines between event vertices `M.fetched` and `M.done`. However, the implementation mapping requires one additional piece of information. Notice that the state vertices `A.fill` and `B.fill` represent the fact that one of the operands has been received and the implementation is waiting for the other. When both operands have been received, the implementation will go ahead and compute the bitwise-or. So in the cross-product construction, we want to invalidate the cross product of vertices `A.fill` and `B.fill`. With this additional information, the composition results in the control graph shown in Figure 10. The resultant control graph captures all possible orderings and arbitrary number of wait cycles. In the top path, the `A` operand was received before the `B` operand. In the bottom path, the `B` operand was received before the `A` operand. In the middle path, both source operands were received simultaneously. The result of the bitwise-or operation is available in the final state vertex.

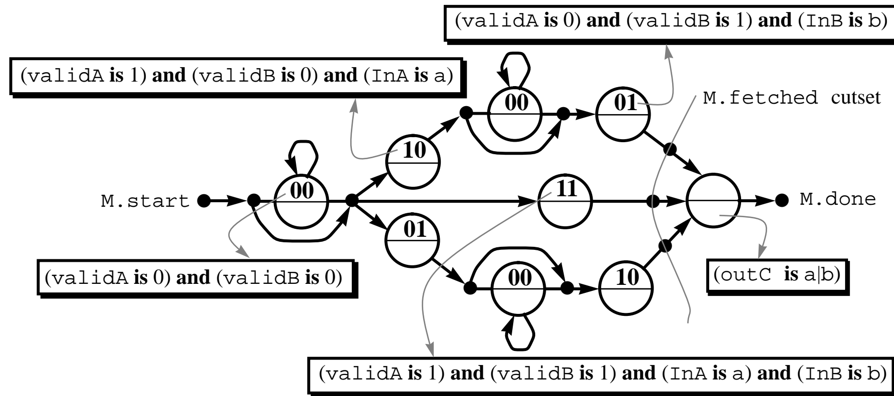

Figure 10. Trajectory Assertion for bitwise-or operation.

The resultant control graph serves as the trajectory assertion. In general the trajectory specification consists of a set of trajectory assertions. Each trajectory assertion can be represented as a tuple: $T = \langle V, U, E, s, t, \sigma_a, \sigma_c \rangle$, where

- $\langle V, U, E, s, t \rangle$ is a control graph.
- σ_a labels state vertices with antecedent node formulas.
- σ_c labels state vertices with consequent node formulas.

The trajectory assertions express the system behavior relative to a path in a control graph. The antecedent labelling along a path places restrictions on the inputs and internal states of the circuit. The consequent labelling specifies the set of allowed circuit responses along the path.

Trajectory assertions can be classified into three categories in increasing order of generalization: 1) Single Sequence 2) Acyclic 3) Generalized. The control graph associated with a single sequence trajectory assertion has a linear sequence of vertices. An acyclic trajectory assertion has a directed acyclic control graph. And the generalized

trajectory assertion has any arbitrary control graph.

4. Verification

The main concern in the verification task is that representing the next-state function or relation for large systems is not computationally feasible. Therefore, we use Symbolic Trajectory Evaluation (STE) which evaluates the next-state function on-the-fly and only for that part of the circuit required by the trajectory assertion. STE uses a partially ordered system model. In the past, STE has been used to verify only single sequence trajectory assertions or single sequences augmented with simple loops. We extended STE to deal with acyclic and generalized trajectory assertions.

4.1. Partially Ordered System Model

The logic set is extended to $\{0, 1, X\}$. The logic value X denotes an unknown and possibly indeterminate logic value. The logic values are assumed to have a partial order with logic X lower in the partial order than both logic 0 and 1 as shown in Figure 11.

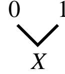

Figure 11. Partial Order for logic values 0,1,*X*.

Assume that $\tilde{N}$ is the set of node assignments in the partially ordered system. Therefore $\tilde{N} = \{0, 1, X\}^m \cup \top$ where $m = |N_v|$. The elements of the set $\tilde{N}$ define a complete lattice $[\tilde{N}, \sqsubseteq]$ with X^m as the bottom element and $\top$ as an artificial top element. The partial order in the lattice is a pointwise extension of the partial order shown in Figure 11. The symbols $\sqcup$ and $\sqcap$ denote the least upper bound and greatest lower bound operations respectively.

In the partial order system, node formulas are represented by an element of the lattice. The restricted form of the node formula ensures that it can be represented by a lattice element. The lattice element corresponding to a node formula F, written $\mathcal{L}at(F)$, is defined recursively:

- $\mathcal{L}at(n_i \textbf{ is } 0)$ is an element of the lattice with i^{th} position in the element being 0 and the rest of the positions being X's. $\mathcal{L}at(n_i \textbf{ is } 1)$ is an element of the lattice with the ith position being 1 and the rest of the positions being X's.
- $\mathcal{L}at(F_1 \textbf{ and } F_2) = \mathcal{L}at(F_1) \sqcup \mathcal{L}at(F_2)$.
- $\mathcal{L}at(0 \rightarrow F) = X^m$ and $\mathcal{L}at(1 \rightarrow F) = \mathcal{L}at(F)$.

4.2. Extensions for acyclic trajectory assertions.

An acyclic trajectory assertion can be verified by enumerating all the paths from source to sink and using STE separately on each path. However, there can be an exponential number of paths in a graph. We use path variables to encode the graph and verify all paths in one single run of the simulator. STE has been extended with a conditional simulation capability. Conditional simulation allows STE to advance to the next-time instant under some specified Boolean condition. Under the condition, the simulator advances to the next-time instant. Under the complement of the condition, the simulator maintains the previous state of the circuit. The trajectory assertions are

symbolically encoded and STE conditionally evaluates the next-state function for each vertex in topological order. As an example, assume an acyclic trajectory assertion of the form shown in Figure 12. The path variables shown in the figure are introduced to encode the graph. For every vertex w with an outdegree of $d(w)$, $\lceil log(d(w)) \rceil$ number of variables are introduced to encode the edges. The conditions are accumulated at the vertices as shown. An expression at a vertex specifies all possible conditions under which the vertex can be reached from the source. The simulator can now compute the vertices in topological order with the vertex expressions being used as constraints for computing the excitation function. The number of additional path variables introduced is $\sum_{w \in W} \lceil log(d(w)) \rceil$, where $W = V \cup U$.

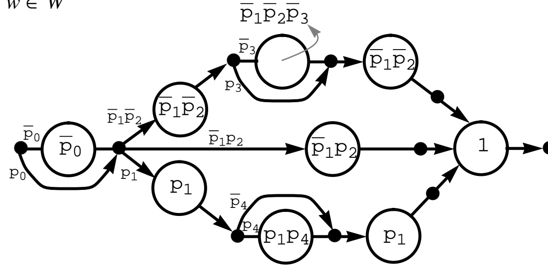

Figure 12. Symbolic encoding of acyclic trajectory assertions.

4.3. Extensions for generalized trajectory assertions.

Cycles in trajectory assertions can be dealt by performing a greatest fixed point computation[4]. Therefore, we can deal with generalized trajectory assertions. We identify the strongly connected components, obtain the corresponding acyclic component graph, and encode the acyclic component graph. The strongly connected components require a fixed point computation and are dealt with in a recursive manner. As an example, the graph in Figure 12 is the acyclic component graph of the trajectory assertion in Figure 10. Note that an exact greatest fixed point computation in a cycle requires introducing new variables for each iteration of the cycle until a fixed point is reached. The number of path variables introduced is no longer defined solely by the trajectory assertion. Therefore, for each iteration of the cycle we quantify out the path variables introduced for the cycle and reuse them in the next iteration. Quantification requires a greatest lower bound operation on the lattice. The greatest lower bound computation can lead to a pessimistic verification. A pessimistic verification can generate false negatives (correct circuit is reported to be incorrect) but will never generate a false positive (incorrect circuit is reported to be correct). In effect STE computes a conservative approximation of the reachability set.

5. Verification of a Superscalar Processor

We are testing our methodology on a superscalar processor. The processor is an implementation of the PowerPC architecture. The particular design is a fully functional prototype of the processor found in IBM's AS/400 Advanced 36 computer[12].

Presently, work has concentrated on verifying arithmetic and logical instructions with two source register operands and one target register operand in the fixed point unit

(FXU). The FXU has three pipeline stages, dispatch, decode and execute. Most of our attention has focussed on verifying the decode stage because this is where the bulk of the control logic resides. The decode stage is responsible for obtaining the source operands from the register file and reserving the target operand register. An instruction might stall in the decode stage because the operands are not immediately available or because the execute stage is busy processing another instruction. If the execute stage has a valid instruction, then the FXU can bypass the register file and forward the target operand data in the execute stage to the decode stage. This occurs when the source register address of the instruction in the decode stage is the same as the target register address of the instruction in the execute stage.

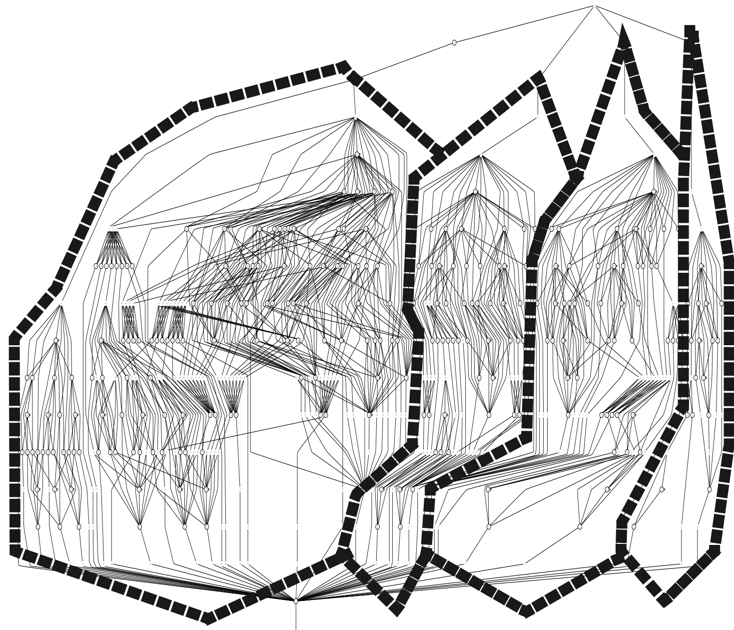

Figure 13. Trajectory Assertion for a class of instructions in the FXU.

We specified the mapping to capture these implementation details. The main machine defined the dispatch, decode and execute stages. A total of 14 map machines described the mapping for simple abstract formulas. Our tool took the abstract specification and mapping and generated the resultant trajectory assertion shown in Figure 13. The trajectory assertion corresponds to all possible cases that can arise due to interactions between the map machines. The control graph can be divided into 4 vertical slices as shown by the thick, dotted lines in the figure. We will number these slices 1 to 4 from left to right. Slice 1 is the most complex of all slices and corresponds to the case where neither of the source operands are bypassed. This slice has to deal with all possible arrival orders of the source operands from the register file. Slices 2 and 3 correspond to the case where one of the source operands is being bypassed. Slice 4 is the simplest of all slices and corresponds to the case where both source operands are bypassed.

The trajectory assertion has 261 state vertices, 57 cycles that require a fixed point com-

putation and 4210 paths in the corresponding acyclic component graph. It would be computationally infeasible to enumerate all paths and use STE separately on each path. Instead our tool encoded the paths using 410 path variables. STE verified the entire trajectory assertion in a single verification run. We took the OR instruction as a representative of this class of instructions. The verification of the OR instruction took 3 hours of CPU time and 70 Meg of memory on an IBM Power Series 850, 133MHz 604.

6. Conclusions

We have presented a methodology for verification of systems that have a simple high-level specification but have implementations that exhibit nondeterministic behavior. The verification task verifies each individual operation. The theoretical foundation behind the methodology ensures that the operations can be stitched together to create arbitrary execution sequences. STE has been enhanced to deal with greater amounts of nondeterminism. Our work on the fixed point unit in the PowerPC architecture indicates that this is a promising approach for the verification of processors.

References

[1] R. E. Bryant, D. L. Beatty and C. J. H. Seger, "Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation," *28th Design Automation Conference*, pp. 397-402, June 1991.

[2] D. L. Beatty, "A Methodology for Formal Hardware Verification with Application to Microprocessors," *PhD Thesis, published as technical report CMU-CS-93-190, School of Computer Science, Carnegie Mellon University*, August 1993.

[3] D. L. Beatty and R. E. Bryant, "Formally Verifying a Microprocessor Using a Simulation Methodology," *31st Design Automation Conference*, pp. 596-602, June 1994.

[4] C. J. H. Seger and R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Formal Methods in System Design 6*, pp. 147-189, 1995.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan and D. L. Dill, "Sequential Circuit Verification Using Symbolic Model Checking," *27th Design Automation Conference*, pp. 46-51, June 1990.

[6] K. L. McMillan, "Symbolic Model Checking," Kluwer Academic Publishers, 1993.

[7] R. P. Kurshan, "Analysis of Discrete Event Coordination," *Lecture Notes in Computer Science 430*, pp. 414-453, 1990.

[8] R. P. Kurshan, "Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach," *Princeton University Press*, 1994.

[9] W. A. Hunt, "FM8501: A Verified Microprocessor," *Lecture Notes in Artificial Intelligence 795,* 1994.

[10] T. K. Miller III, B. L. Bhuva, R. L. Barnes, J.-C. Duh, H.-B. Lin and D. E. Van den Bout, "The Hector Microprocessor," *International Conference on Computer Design*, pp. 406-411, 1986.

[11] M. Srivas and M. Bickford, "Formal Verification of a Pipelined Microprocessor," *IEEE software 7(5)*, pp. 52-64, September 1990.

[12] C. May, E. Silha, R. Simpson and H. Warren, "The PowerPC Architecture: A Specification for a New Family of RISC Processors," *Morgan Kaufmann Publishers*, 1994.

[13] J. R. Burch and D. L. Dill, "Automatic Verification of Pipelined Microprocessor Control," *Lecture Notes in Computer Science, Computer Aided Verification, 6th International Conference, CAV 94*, pp. 68-80, 1994.