

# Architecture Validation for Processors

Richard C. Ho, C. Han Yang, Mark A. Horowitz and David L. Dill

Computer Systems Laboratory,  
Stanford University,  
Stanford, CA 94305.

## Abstract

Modern, high performance microprocessors are extremely complex machines which require substantial validation effort to ensure functional correctness prior to tapeout. Generating the corner cases to test these designs is a mostly manual process, where completion is hard to judge. Experience shows that the errors that are caught late in the design, many post-silicon, are interactions between different components in very improbable corner case situations. In this paper we present a technique that targets such error-causing interactions by automatically generating test vectors that will cause the processor to exercise *all* transitions of the control logic in simulation. We use techniques from formal verification to derive transition tours of a fully enumerated state graph of the control logic of the processor. Our system works from a Verilog description of the original machine and is currently being used to validate an embedded dual-issue processor in the node controller of the Stanford FLASH Multiprocessor. Modeling the processor control results in 200K states and an 8M instruction trace to check all transitions of control arcs.

## 1 Introduction

One of the hardest tasks facing a microprocessor design team and one which requires large amounts of manpower and resources is the job of verifying that a design is bug-free prior to tapeout. Modern microprocessors contain many architectural features designed to improve performance, including branch prediction, speculative execution, lock-up free caches, dynamic scheduling, superscalar execution [Joh91] and others. All these features add complexity to the machine and interact with each other making the validation task more difficult. Currently, this validation is done through simulation, often using hardware-assist [Gat94] to reduce runtime. The key in simulation-based validation is to create tests that exercise a bug, i.e. make the error apparent to the user. Today, tests are either hand-written and directed towards particular test cases or are randomly-generated in hopes of probabilistically reaching untested situations. Both of these methods fail to provide a measurable degree of confidence that a complex design is

adequately tested. Many bugs are not discovered until silicon is produced or worse, after the product has shipped.

The bugs which are hardest to find are those that result from extremely improbable events, often the product of corner cases in several different components of a complex design. A typical example of a bug in this “multiple events” class is one in the MIPS R4000PC/SC rev 2.2 processor. The error occurs when an instruction sequence contains a load instruction which causes a data cache miss, followed by a jump instruction with its delay slot on an unmapped page. When the TLB miss exception is taken, the jump address is erroneously used instead of the page miss exception vector. This type of bug is hard to find using hand-written tests because test writers cannot guarantee that every possible interaction is exercised. Random testing might find this case, but each of the conditions is so improbable that finding an error that occurs at the conjunction of these cases requires a prohibitively large number of simulation cycles.

In fact, the published errata from the MIPS R4000 [Mips94] and the tales of bugs from companies, show that the major cause of errors that slip through existing verification methods consist of exactly these interactions between parts of a design in corner case situations. Table 1.1 shows the breakdown of bugs for the MIPS

Bug Class	Number of Bugs	% of Total
Pipeline/Datapath ONLY bugs	3	6.5%
Single Control Logic Bugs	17	37.0%
Multiple Event Bugs	26	56.5%
Total Reported Errata	46	100.0%

**Table 1.1. Classification of MIPS R4000 Errata**

machine classified according to the parts of the design that interacted to cause the error: pipeline/datapath only, single control logic error or multiple interaction error.

The methodology presented in this paper brings together formal methods with simulation to tackle this problem of testing a microprocessor design in all its corner

case situations. It does so by systematically generating test vectors that will exhaustively simulate all possible interactions of the sub-units. Formal methods are used to generate a complete state graph of the control logic, from which test vectors are automatically generated which cause the simulation to pass through all possible control state transitions. Using the complete set of vectors maximizes the probability of finding errors in the smallest amount of simulation time.

This technique differs from other microprocessor verification techniques such as [BeB94], [BuD94] and [BhD94] by addressing the task of verifying actual microprocessor designs including advanced architectural features. It is currently being used to validate the design of a statically-scheduled, dual-issue superscalar RISC microprocessor core, which is part of the Stanford FLASH Multiprocessor project [KOH+94]. Preliminary runs of the tool on a “tested” design have uncovered a number of “multiple event” class errors, which we find very encouraging.

The next section gives a brief overview of the processor design we are using as our first test case for this method. While it is simple by today’s standards, it is a fully pipelined processor with an advanced memory system. It also has a large number of interfaces to functional units integrated on the same chip, making it a challenging validation problem. Section 3 then begins by describing our test generation method and explains how we use the actual machine description to generate the desired tests. Generating tests is basically a three step process: translate the Verilog into a FSM description, enumerate the control graph of the machine, and find instruction traces that cause all the arcs in the graph to be exercised. Section 4 then presents some limitations of our current approach, and areas we are working on improving.

## 2 The Stanford FLASH Protocol Processor

As part of a multiprocessor design effort at Stanford, a custom memory and interconnect controller is being designed. This controller (MAGIC) is unusual since it contains a general purpose processor. The processor runs protocol code to enable the machine to efficiently support cache coherent shared memory and high performance message passing [KOH+94]. The Protocol Processor (PP) works in a control pipeline, shown in Figure 2.1., getting new tasks from the Inbox and sending completed tasks to the Outbox. It is a DLX-based, statically scheduled, dual-issue RISC processor core with instruction and data caches and special instructions to communicate with the other functional units on MAGIC. In many ways, the PP is a simple processor since it does not support virtual memory or recoverable exceptions. However, it does contain a high performance memory system, squashing branches and a

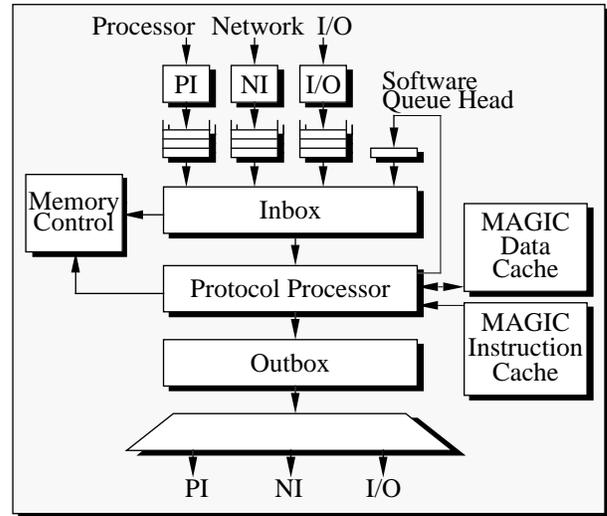


Figure 2.1. MAGIC Macro Control Pipeline

number of external interfaces which make it difficult to check.

The data cache is two-way set associative and uses a “fill-before-spill” refill strategy to reduce latency on a miss which requires a write-back of a dirty line. The dirty line that is to be replaced gets copied to a temporary “spill buffer” so that the cache controller can fill the cache line from memory and restart the stalled processor first. To reduce the latency of a miss even further, the PP does a “critical-word-first” restart, where the stalled processor resumes execution after receiving the data-word that it missed on rather than waiting for the entire line to be retrieved from memory. Both of these operations can cause stalls to occur if further instructions need the hardware resources while the previous operation has not completed. Split stores are performed to the data cache to reduce overall store latency. The data tag probe is executed in one cycle and the store executed in a following one. If the store is followed by load instructions to different cache lines, the store is delayed to allow the loads to execute first, thereby completing ahead of the store. If the load is to the same cache line or another store is executed, the processor is stalled on a “conflict stall” while the first store is completed.

Extensions to the instruction set to communicate with other functional units introduce new stall conditions if one of the other units on the MAGIC chip is not ready. For example, there is an instruction called `send`, that communicates with the Outbox. If the Outbox is not ready to accept the `send` from the PP, then the PP must stall when it reaches execution of the `send` in its pipeline.

The real advantage of this machine is that it is not a “toy” example nor is its validation an academic exercise. A large number of people are counting on this chip to

work for the FLASH project. Thus we can compare how this technique works on well designed and well tested logic.

## 2.1 Protocol Processor Bugs

We have begun using automatically generated test vectors to drive the RTL model and compare the result to an instruction-level simulator. Already we have discovered several non-trivial bugs that were not uncovered using hand-written or randomly generated test vectors. The test vectors were run on the PP Verilog model after it had been partitioned for synthesis and was considered fairly mature. All bugs that were found using other methods were found using this technique. In addition, Table 2.1 shows some bugs found using the generated vectors but not (yet) found by other methods over the course of a couple of months of testing and debugging both software and the design. During this time, we have focused on modeling the memory system and the stall machine.

Bug #5 is illustrative of the improbable interactions that are involved in some bugs which make them hard to detect. The situation leading to the bug was a load that missed in the data cache followed by any other load or store instruction. The first load required retrieval of a line

from memory to cache. With critical-word-first restart, the first word returned from memory was driven onto a bus (Membus) leading to the register file. The bug existed as a glitch on the signal that indicated valid data on the Membus, as shown in Figure 2.2. The glitch was caused by the

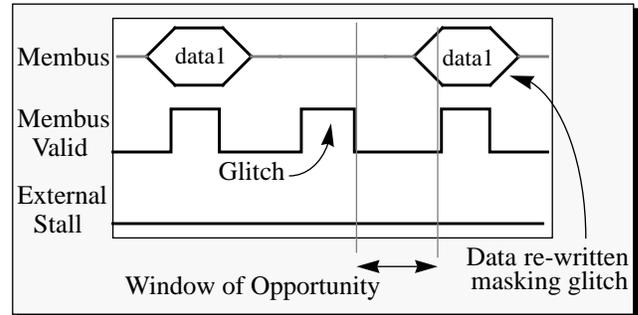


Figure 2.2. Bug #5 Timing Diagram (Glitch Masked)

presence of a load or store instruction in the pipe following the load that missed and it occurred after the critical word was driven, thereby overwriting it with potential garbage because the bus is at high impedance at this time. However, the logic which implemented the refill was still erroneously implementing an older restart policy. So, it

Bug	Description (Summary of Bug followed by Explanation)
1	Interface miscommunication between PP's cache controller and the Memory Controller. Qualification of an interface signal was needed, but the two units thought that the other would perform it. The bug manifested itself as incorrect data being returned to the I-Cache.
2	Latch not qualified on all stall conditions and lost data. On a simultaneous I & D Cache miss, the latch holding the data that was to be returned after the D-Cache refill was not qualified on the I-Stall and lost its data by the time the I-Cache miss was serviced.
3	Cache conflict stall can cause wrong address to be used on the stalled load. The address used in the load of a conflict stall was not held during the stall. If there was no following instruction that used the address bus of the cache (any load/store instruction), then the correct address from the load remained. However, if the load in the conflict stall was followed by another load/store instruction, then the address of the following load/store was erroneously used.
4	I-Stall fix-up cycle lost if I-Stall condition occurs during Mem-Stall. The I-Cache refill machine takes a cycle to restore the correct values to the instruction registers after an I-Stall. However, it was not qualified on MemStall, so was lost if the I-Stall condition arose after MemStall was asserted. This can happen if a <code>switch</code> or <code>send</code> is executing in the stalled instruction and the external unit (Inbox or Outbox) signals the PP to wait.
5	Glitch on bus valid signal allows Z values to be latched on a load that missed followed by any other load/store instruction interrupted by an external stall condition. This bug is explained in detail in the text.
6	Cache conflict stall with D-Cache hit and simultaneous I-stall results in stale data being loaded. A cache conflict stall occurs because of the split store operation. When the address of the load following a store is the same as the store, a conflict stall is taken to write out the store data before loading it. When there is a simultaneous I-stall caused by an external condition, the load receives the stale data instead of the newly written data.

Table 2.1. Synopsis of Discovered Bugs

then drove the required data onto Membus a second time, giving the correct result to the register file. This in itself is a performance bug which our result comparison does not find. The correctness bug exists only if an external stall condition arose between the time of the glitch and the second write, preventing the second write from occurring, leaving garbage in the register file, as shown in the timing diagram of Figure 2.2. In actual hardware, it would have

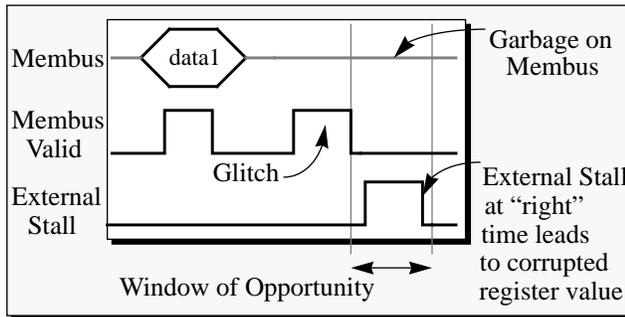


Figure 2.3. Bug #5 Timing Diagram (Garbage written)

occasionally shown up as corrupted data and its reproducibility would have been limited because of the requirement of a stall arising from another asynchronously operating unit. Easily generating this rare interaction is the advantage of our method. The next section describes how we generate these test vectors in more detail.

### 3 Methodology Overview

While the designers spend a large amount of time considering improbable states that the machine can enter and generating tests for them, these tests are based on what they believe the corner cases will be. A common bug scenario is that a corner case situation was overlooked by a designer both in the design and in the test generation. This can also happen with an independent test writer if the corner case is particularly complex. Since the hardware description of the machine is complete, it too must contain information about the improbable states, information that is likely to be more complete than the things that a person remembers. The goal of our method is to extract this information from the design to generate test vectors. We do this in an automated fashion using a three step process which is shown in Figure 3.1 The first step (the oval marked as 1 in the figure) is a translator that converts the HDL representation to a *Finite State Machine* (FSM) representation. This step captures the control logic of the design as a set of interacting FSMs. Step two performs a *full state enumeration* of the FSMs to find all reachable states from reset. It produces a complete state graph which contains every state and every transition edge that the hardware model can get to. This state graph accurately predicts all behaviors of the design since it is derived directly from the HDL model. Step three takes the complete state graph and generates vectors that will cause the processor being tested to take a *transition tour* of its state graph. A transition tour is one in which every edge of a directed graph is traversed at

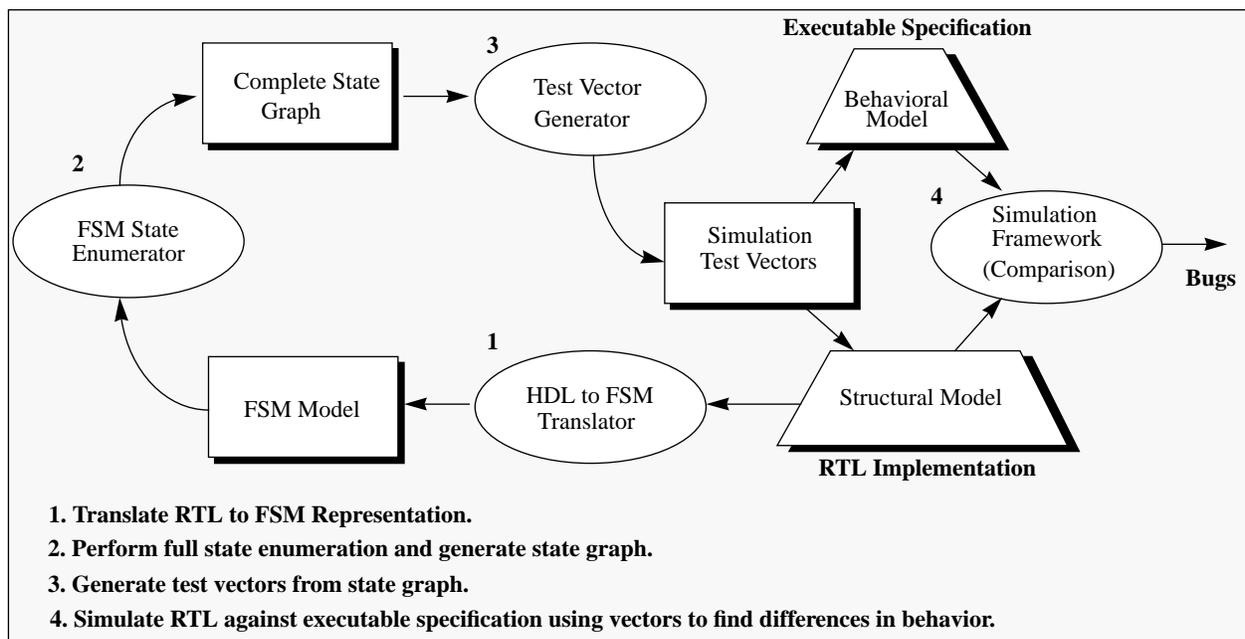


Figure 3.1. Methodology Overview

least once. Hence every control transition edge in the design will be simulated at least once with the vectors. The vectors are produced from the transition tour by a *transition condition mapping*, which simply takes the condition that causes a transition to occur in the FSM model and produces a test vector that matches the condition. The parts of the vector that do not impact the control logic FSMs, for example the data value and the precise operation type, are set randomly.

Most of the steps are automated. The most difficult step is the first one, the extraction of the FSM model. Here the designer needs to initially annotate the HDL model to aid the translator in finding the control logic. This needs only be done once with updates whenever the control logic changes. After that, the framework can proceed without user intervention until errors are found in the simulation and human analysis of the bug is required. The following sections describe the three steps in more detail.

### 3.1 HDL Translator

One of the reasons that formal methods are not more commonly applied to processor design is the difficulty of converting a design description, usually in the form of an HDL such as Verilog, into the form used by the formal technique. Additionally, during the design phase, changes are frequent and numerous. If the conversion process is slow or laborious, changes will not be incorporated as frequently or as rapidly as needed to keep up with design changes that may introduce bugs. Hence our methodology derives all models directly from Verilog using a translator to the language of our state enumeration tool, Synchronous Mur $\phi$ , which is described briefly in the next section.

Since the methodology aims at representing the *implementation* model of the design as FSM, it is sufficient that a *stylized synthesizable* subset of Verilog is translated. With this subset, translation is mostly a one-to-one syntactic correspondence between Verilog and Synchronous Mur $\phi$ . This is made possible because the Verilog model of concurrency, in which the implicit clock is advanced only when all variables are stable, can be easily mapped into the Synchronous Mur $\phi$  model of concurrency, which has an explicit separation of state and non-state variables and the implicit clock updates state variables only.<sup>1</sup> To allow for Verilog that does not fully conform to the stylized subset (like error and diagnostic code), comment embedded directives can be used to turn translation on and off.

1. The only difficulty in semantics involves Verilog variables that hold state across cycles. The concurrency model of Verilog makes these latches implicit in the stylized code. However, they need to be explicit in Synchronous Mur $\phi$ , so the translator must analyze for latches and convert them to explicit state variables.

The hard problem in this translation is the question of what should be considered state. Since the program has problems distinguishing data storage from state storage, we rely on user annotations in the code to guide this process, both to indicate which bits are state bits, and to specify the number of distinguished cases. A good example of the need for a user to specify distinguished cases is for instructions in a pipelined processor. Since the processor is pipelined, part of its state is the set of instructions currently in execution. With a 5 stage pipeline and 100 different instructions, the number of possible states is large. Yet from the control's perspective many instruction executions look the same and could be collapsed into a single *instruction class*. Generally from the controller's perspective, there are only a few instruction classes, which greatly reduces the state space that needs to be traversed. To provide a better feeling for the kind of information that needs to be provided, the next paragraph describes the annotations used for the protocol processor.

The annotations needed for the PP were relatively small, since the design was already partitioned into datapath and control for synthesis. The datapath contributions to the state machines are reduced to a few distinguished cases based on how the datapath value affects the processor state. In the PP, there are two datapath elements that have an impact on control logic: memory addresses and instructions in the pipeline. Memory addresses, including the program counter (PC), are reduced to a single bit which indicate whether the address it represents causes a hit or a miss in the cache. Instructions are reduced to five instruction classes, shown in Table 3.1. These classes are the

Instruction Class	Effect on Control Logic
ALU	Has no effect since there are no exceptions in the PP.
LD	Execution of a load can cause transitions in load/store FSMs.
SD	Execution of a store can cause transitions in load/store FSMs.
SWITCH	A switch instruction executed while the Inbox is not ready causes a pipeline stall.
SEND	A send instruction executed while the Outbox is not ready causes a pipeline stall.

**Table 3.1. PP Instruction Classes**

ones required to model the memory system and stall machine. We have not yet modeled squashing branches, but when we do, there will be additional instruction classes for control transfer instructions. At the moment, branches only impact the control logic by causing instruc-

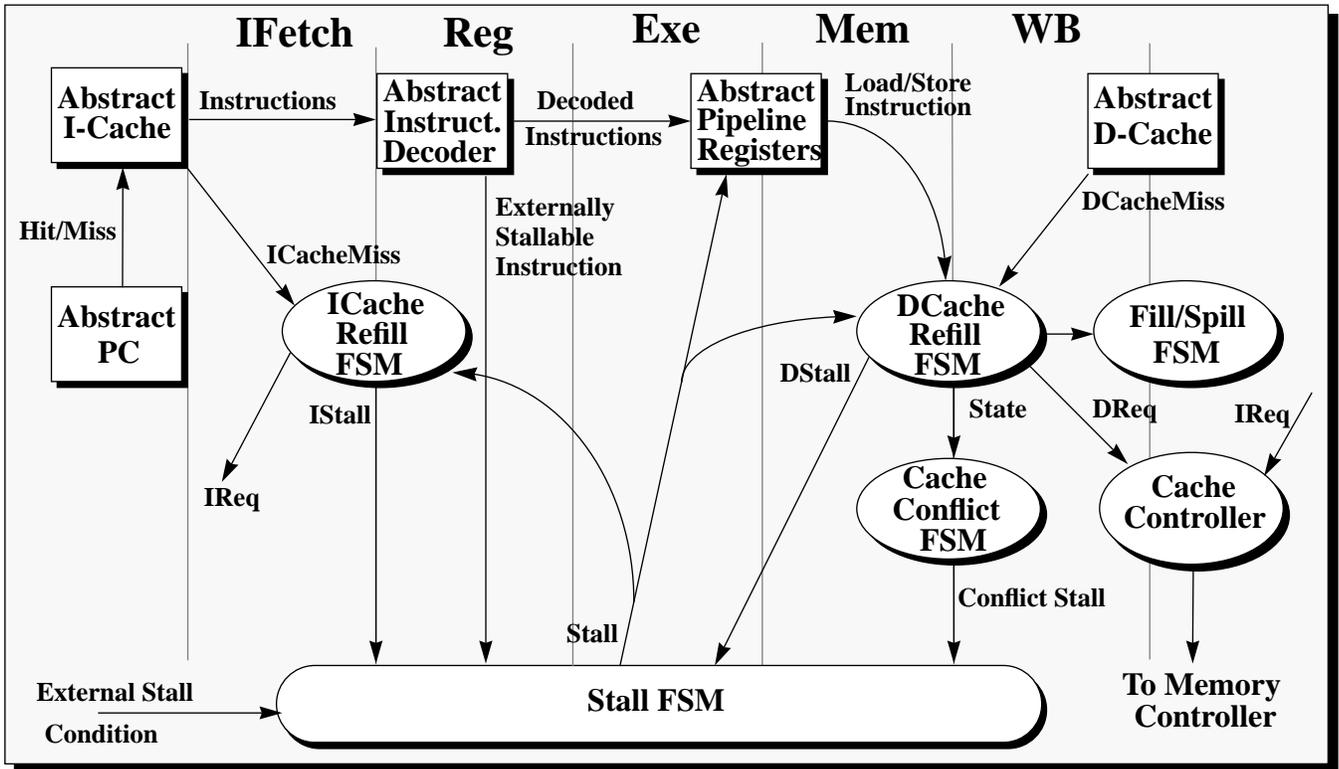


Figure 3.2. FSM Representation of the PP with Modeled Abstraction

tion cache misses, so they are included in the ALU instruction class.

With these abstractions, annotating the Verilog model consists of delimiting the sections of control logic that correspond to state machines and any logic that feeds the state machines. Logic that takes state values and drives the datapath is not required and is left outside the delimited areas. Out of a total of 2727 lines of Verilog code in the control sections, 581 lines were included in the delimited areas. Datapath and control signals from other units that feed into the state machines are modeled abstractly. For the PP, abstract models of the PC, the on-chip memory controller, pipeline registers, Inbox, Outbox and both caches are needed, as shown in Figure 3.2. Since these abstract models form the interface to the control logic, they must cause the control logic to exercise every transition arc. This is done by making these abstract models try every combination of values. With Synchronous Mur $\phi$ , this is done by specifying the models as blocks that can non-deterministically choose one of several possible actions. During the state enumeration, all possible choices of actions are permuted for each state, resulting in the discovery of all reachable states, no matter how improbable a sequence of interactions is needed to reach it. The next section describes how state enumeration is performed and the results of enumerating the control states of the PP.

### 3.2 State Enumeration

After modeling the control logic as FSMs, the next step is to generate a full state graph of the model. For this, we use a formal verification tool developed at Stanford called Synchronous Mur $\phi$ , which is an extension of Mur $\phi$  [DDH+92]. Synchronous Mur $\phi$  finds all reachable states of the model by doing breadth-first search starting with the given reset state. As a new state is found, the choice of actions that caused the transition from the current state to the new state becomes the edge of the state graph. Although more than one permutation of actions can cause the same transition from one state to another, only one is recorded to become part of the state graph. This eliminates the redundant work of repeatedly following the same arcs but can lead to problems as described in Section 4.

The hard problem with using explicit state enumeration is dealing with state explosion. Our use of distinguishing values helps a lot by reducing datapath elements to just a handful of bits. In addition, the interfaces to the control logic are modeled abstractly without requiring additional state. Synchronous Mur $\phi$  ensures that all possible combinations of values are tried without requiring us to explicitly keep track of this. However, we believe that the most likely reason that the number of states in the processor control logic is manageable is that the different FSM that

comprise the model have a sort of interlock between them. For example, once a data cache refill starts, the instruction cache refill state machine must wait to handle its refill since there is only one path to the memory controller. It appears that the mutual stalling of FSM prevents the exponential explosion in states that would be expected based on the number of state bits required by the model. Table 3.2

Number of States	229,571
Number of bits per State	98
Execution Time (on DecStation 5000/240)	18,307 cpu secs.
Memory Requirement	34 MB
Number of Edges in State Graph	1,172,848

**Table 3.2. State Enumeration Statistics**

shows some statistics of the state enumeration of the PP FSM model. The total number of states is only of the order of  $2^{18}$  rather than the  $2^{98}$  possible states suggested by the number of bits for the state encoding. This situation will worsen when we include squashing branches into the model, but we are still hopeful that the total number of control states will remain manageable. Once the full state graph is available, test vectors can be generated from this. The next section talks in more detail about how this is done.

```

GenerateTours () {
    state = InitialState;
    open output file to write tour;
    do {
        do {
            /* Depth first traversal generating a vector for every edge traversed. States can be
               visited multiple times as long as there is an untraversed edge from that state. */
            state = TraverseDFS (state);

            /* When DFS cannot find a state with untraversed edges, perform a breadth first
               search looking for any state that has an untraversed edge. If found, generate the vectors
               to reach this state from the point the DFS stopped at. */
            state = TraverseBFS (state);
        } while (we can find a state with an untraversed edge) &&
                (number of instr. generated <= MAX instructions per file);
        close output file and open new output file to write new tour;

        /* Explore phase - check whole graph for any remaining untraversed edges. */
        state = TraverseBFS(InitialState);
    } while (there exists a state with an untraversed edge);
    Remove empty last output file;
}

```

### 3.3 Test Vector Generation

Given a directed graph of the control state, we would like to generate test vectors that cause the simulation to exercise every arc in the graph. A series of arc transitions that traverses every arc at least once is known as a transition tour. A transition tour that traverses every arc *exactly* once is known as a *Euler Tour*. However, a Euler Tour can only be found for symmetric graphs that are strongly-connected. The general problem of finding a transition tour in a non-symmetric strongly-connected graph is called the Chinese Postman Problem [EJ72] and is most frequently encountered in the field of protocol conformance testing, which has developed algorithms for finding such tours in polynomial time [Hol91].

However, this validation methodology does not require a strict transition tour and in fact, there are good reasons not to use a single transition tour. To make concurrent simulation possible and to limit the simulation time needed to reach any bugs found, we break up the transition tour into smaller components that all start from the reset state. In this case, the requirement is that the union of all the arc transitions taken by all the tour components cover all the arcs in the state graph. Additional requirements are that we try to avoid operations which are costly in simulation, namely backtracking and setting the system to a particular state. Traversing an edge multiple times is cheap in simulation, so this is done to avoid backtracking. Where possible, we try to proceed in a depth-first manner, since this

**Figure 3.3. Tour Generation Algorithm**

translates naturally to the simulator advancing cycle-by-cycle. We use a “greedy” algorithm which performs a depth-first-search (DFS) of the graph, using each transition edge as part of the tour. This proceeds until an untraversed edge cannot be found. At this point, the algorithm goes into an *explore* phase looking in a breadth-first manner for an untraversed edge but without adding edges to the tour. Once found, the shortest path from the point where the DFS stopped to the untraversed edge is added to the tour and DFS can continue. If such a node cannot be found, the algorithm starts a new tour starting from the reset state. This continues until no new tours can be found even from reset. The algorithm used to create this set of partial-tours is shown in Figure 3.3 as pseudo-code.

Converting from a transition tour to test vectors requires that the simulation be driven to take the transitions specified in the tour. For processors, there are two classes of stimuli that affect control: the instruction stream and input signals from external sources. As long as both these sources of stimuli are controlled in the simulation, the model will take the same transition arcs as the transition tour.

As described in Section 3.2, the processor FSM model is interfaced to abstract models of the datapath and other units of MAGIC. In the enumeration of the state graph, state transitions were labelled with the choice of actions taken by each abstract block. So, to generate the vectors which represent the transition tour, we forcibly take control of the signals in the simulator which interface to the control logic and make them match the choice of the abstract blocks. For Verilog, this is done by writing a set of “force/release” commands to toggle the values of the interface signals. When the simulation is run, these commands are compiled with the model and cause the interface signals to transition at the times specified by the transition tour. The correspondence between interface signals in the FSM model and actual wires in the simulation is made in the *transition condition mapping*. The second source of stimuli, instructions, come from the abstract model of the instruction cache. The instruction class chosen for each transition is encoded like any other abstract model choice. When the transition tour is traversed to generate the test, a random instruction from the class is chosen along with random data.

Table 3.3 shows the results of applying this method to the PP state graph described in the previous section. It shows that simulating the complete set of vectors is achievable and on average a modest number of instructions (7) is needed to test each arc. In addition, breaking up traces into smaller components does not add much overhead, which is fortunate since it is extremely helpful in reducing the time needed to rerun a simulation to reach a bug.

	With no limit	With trace limit (10,000 instructions)
Number of Traces Generated	1,296	1,296
Total number of edge traversals generated	21,200,173	21,252,235
Total number of instructions generated	8,521,468	8,557,660
Generation time (on DecStation 5000/240)	161,159 cpu secs.	193,330 cpu secs.
Estimated simulation time @ 100Hz (total)	58.9 hours	59.0 hours
Longest Single Trace	21,197,977	144,520 edges
Estimated simulation time @ 100Hz (longest trace)	58.9 hours	24 mins.

**Table 3.3. Test Vector Generation Statistics**

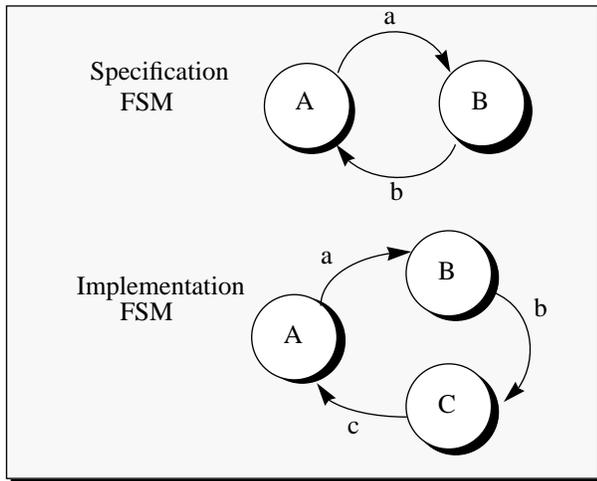
It might seem strange at first that the same number of traces were generated in both cases. It turns out that the PP model has numerous edges in the graph that can only be reached from reset. These edges represent different initial conditions for the inputs to the model and the traces needed to reach these edges cannot be combined with others. This gives us the lower bound on the number of traces needed to cover all edges in the model (1,296 for the PP model). Without an instruction limit, over 99% of the instructions were generated in the first trace and the remaining 1295 short traces were needed to cover different initial conditions. With the instruction limit, many of the edges in the state graph that were originally covered by trace 1 are spread out among later traces. A total of 853 traces were terminated due to the 10,000 instruction limit which suggests that it took 854 traces to cover most of the edges originally covered in the first trace. The remaining 442 traces covered the remaining initial conditions.

## 4 Issues and Future Work

While this test generation method holds a lot of promise, designers should be aware of some caveats. The ability of this technique to detect bugs in the design relies on two things. Firstly, the bugs must manifest as data value differences between the implementation and the specification. Any behavior that is not modeled in the specification cannot be checked for correctness. This leaves open the possibility that performance bugs may be in the design and not detected. The only way to detect such bugs with our result comparison is to make the specification model cycle-accurate. Unfortunately, a common method of making two cycle-accurate models, one simulator and one RTL model,

is to base one on the other, which increases the probability of including the same errors in both models and hence not finding them. In our initial validation of the PP, we did not attempt to find performance bugs.

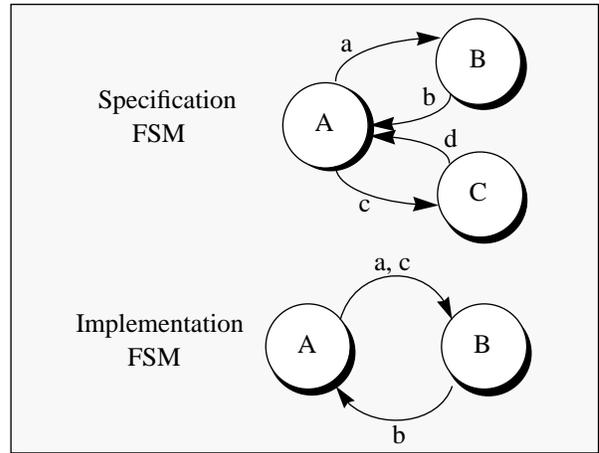
Secondly, the technique tries to expose bugs by exercising all control edges in the state transition graph. By enumerating on the implementation FSM, we capture a class of bugs where the implementation has more behaviors than the specification as demonstrated in Figure 4.1. When



**Figure 4.1. Erroneous FSM Implementation with more Behaviors**

the “c” transition of the implementation is simulated, the difference with the specification is exposed and hopefully the error will be detected. However, there does exist a situation in which this methodology might fail to find a bug as shown in Figure 4.2. For this example, let us assume that the implementation erroneously performs the same state transition for both input “a” and “c”. However, in the state enumeration, each arc is labelled with the first condition leading to a new state, so either “a” or “c” will label the arc depending on which is tried first, but not both. If “a” labels the arc, then the wrong “c” transition will never be exercised to expose the bug. Although we have not done this in our initial validation, this case can be caught by performing the state enumeration on both the implementation FSM and an abstract model of the specification FSM. Another solution would be to change the state enumeration tool to capture all unique transition arcs even if an arc with a different condition already exists between the states.

This validation work is an ongoing part of the FLASH project. The next stage will be modeling squashing branches. This entails adding new instruction classes and an abstract model of the branch outcome determination. Further into the future, we hope to extend this method to other parts of the MAGIC chip, since its applicability is not limited to just processors. Any hardware that can be



**Figure 4.2. Erroneous FSM Implementation with fewer Behaviors**

separated into control and datapath sections and which have complex interactions should be amenable to testing of corner cases using this. To avoid state space explosion, we are planning on using the abstract models of the interfaces as a way to modularize the design. For example, from the Outbox control logic (see Figure 2.1), the entire PP looks like a single wire indicating that a SEND instruction was executed. All of the state present in the PP is abstracted to one bit in this case. We expect that some of these abstract models may be too “liberal” resulting in situations that cannot really occur in the real design. If we encounter this, we plan on constraining the abstract models based on the state enumeration of the real unit.

## 5 Related Work

The work of Iwashita *et. al.* [IKN+94] bears similarities to this work in attempting to use an FSM model of micro-processor control logic to generate test programs that are used in simulation. However, Iwashita *et. al.* focus on the pipeline and target test cases of pipeline hazards for test generation. Our work does not target particular test cases but instead aims to enumerate through the large number of improbable conditions in corner cases in the hope that this will maximize the probability of exposing bugs.

The AVPGEN system from Chandra *et. al.* [CIJ+94] has the same goal as our method, namely generation of test programs to exercise the processor model in “interesting” situations to expose bugs. Their use of constraint solving to choose initial data values and register use for instructions has some advantages over our biased-random method of choosing these. However, to generate interesting instruction sequences, they require templates that specify the ordering of instruction classes in the test. These templates, called Symbolic Instruction Graphs, need to be written by designers to stress particular corner cases.

Our method focuses on exercising these corner cases automatically.

Other microprocessor verification techniques based on formal methods, [BeB94], [BuD94] and [BhD94] to name a few, are limited by the difficulty of handling large designs in formal representations. Our method avoids some of these problems by abstracting datapath and allowing simulation to determine whether a result is correct. More importantly, we derive the FSM model directly from Verilog with a translator making it more likely that bugs in the design are modeled and can be exposed.

To some extent, the technique of generating tests based on state enumeration is similar to work in protocol conformance testing. In both, FSM models of the system are used to generate test sequences that try to exercise a model through all possible control state transitions. The biggest difference is that in conformance testing, only the specification is observable. Hence, extra behaviors in the implementation can be missed since the state enumeration is on the specification (see Section 4). This also makes determining what state the implementation is in a major problem which impacts the test vector generation algorithms used [ADL+91].

## 6 Conclusion

We have presented an automatic method of generating test vectors that systematically exercise a design in highly improbable corner case situations. The method generates test vectors starting from a synthesizable Verilog description, eventually leading to transition tours of a fully enumerated state graph of the control logic. When the vectors are run in simulation, they drive the design through all its control transition arcs in the hope of exposing bugs. We have shown that this method is robust enough to handle some of the architectural features now found in high-performance microprocessors by applying it to the Stanford FLASH Protocol Processor where we found some bugs requiring improbable corner cases conditions that had not been uncovered in earlier testing.

## References

- [ADL+91] Alfred V. Aho, Anton D. Dahbura, David Lee and M. Umit Uyar, "An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours", In *IEEE Transactions on Communications*, Vol. 39, No. 11, November 1991.
- [BeB94] Derek L. Beatty and Randal E. Bryant, "Formally Verifying a Microprocessor Using a Simulation Methodology", In *Proceedings of the Design Automation Conference*, June 1994.
- [BuD94] Jerry R. Burch and David L. Dill, "Automatic Verification of Pipelined Microprocessor Control", In *Computer Aided Verification*, June 1994.
- [BhD94] Vishal Bhagwati and Srinivas Devadas, "Automatic Verification of Pipelined Microprocessors", In *Proceedings of the Design Automation Conference*, June 1994.
- [CIJ+94] A.K. Chandra, V.S. Iyengar, R.V. Jawalekar, M.P. Mullen, I. Nair and B.K. Rosen, "Architectural Verification of Processors Using Symbolic Instruction Graphs", In *Proceedings of the International Conference on Computer Design*, October 1994.
- [DDH+92] David L. Dill, Andreas J. Drexler, Alan J. Hu and C. Han Yang, "Protocol Verification as a Hardware Design Aid", In *Proceedings of the International Conference on Computer Design*, October 1992.
- [EJ72] Jack Edmonds and Ellis L. Johnson, "Matching, Euler Tours and the Chinese Postman", In *Mathematical Programming*, Vol 5. pages 88-124, North-Holland Publishing Company, 1973.
- [Gat94] James Gateley, "Logic Emulation Aids Design Process", *ASIC & EDA*, July 1994.
- [Hol91] Gerard J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall Software Series, Englewood Cliffs, NJ, 1991.
- [IKN+94] Hiroaki Iwashita, Satoshi Kowatari, Tsuneo Nakata and Fumiyasu Hirose, "Automatic Test Program Generation for Pipelined Processors", In *Proceedings of the International Conference on Computer Aided Design*, November 1994.
- [Joh91] Mike Johnson, *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.
- [KOH+94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy, "The Stanford FLASH Multiprocessor", In *Proceedings of the International Symposium on Computer Architecture*, June 1994.
- [Mips94] MIPS Technologies Inc., "R4000PC/SC, Processor Revision 2.2 and 3.0 Errata", [http://www.mips.com/HTMLs/R4000\\_PC\\_bug.html](http://www.mips.com/HTMLs/R4000_PC_bug.html).