Validation Coverage Analysis for Complex Digital Designs*

Richard C. Ho and Mark A. Horowitz

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

Abstract

The functional validation of a state-of-the-art digital design is usually performed by simulation of a register-transfer-level model. The degree to which the test-vector suite covers the important tests is known as the coverage of the suite. Previous coverage metrics have relied on measures such as the number of simulated cycles or number of toggles on a circuit node, which are indirect metrics at best. This paper proposes a new method of analyzing coverage based on projecting a minimized control finite-state graph onto control signals for the datapath part of the design to yield a meaningful metric and provide detailed feedback about missing tests. The largest hurdle is state-space explosion. We describe two methods of dealing with this in a practical manner and give results of applying this coverage analysis to parts of the node controller of the Stanford FLASH multiprocessor.

1 Introduction

Functional verification of a circuit is usually done by simulation of a register-transfer-level (RTL) model. The test vectors used can be generated by a variety of means including pseudo-random generators [1], constraint-solvers [2] or hand-written by designers. No matter how the test vectors were generated, it is important that they cause the RTL-model to be exercised in "interesting ways" that will hopefully expose bugs that may be present in the design. In general, this means generating vectors that test corner-cases, simultaneous events and rare control paths, which is a difficult task.

A coverage metric is often used to check that the test-vector suite does an adequate job of exercising the model. There are many ways to calculate coverage; one of the more complete measures is to use state-transition coverage of the control logic [3]. This gives a good picture of how many of the possible interactions in the control logic have actually been tested. However, we have found that many of the tests that are required by this metric turn out to be equivalent and result in an over-pessimistic coverage metric. Instead we propose a technique based on static-analysis of the model that allows us to focus on the control-state interactions that affect the datapath, and remove from the state-graph any variables which can be tested independently. This reduces the number of transition edges that need to be covered while maintaining the quality of the metric. With fewer transition edges and fewer state-variables to consider, we can highlight the important interactions without the cluttering effects of the independent state-variables. In our experience, this yields much better feedback for the designers.

These independent state-variables arise in many large designs because only a subset of all the variables directly control the datapath actions. It is the cross-product of these variables that need to be exercised to test interactions in the datapath. So long as we have found all sequences of such cross-products, we do not need to further consider the independent state-variables, which can then be tested by themselves. We generalize this observation, in Section 2, by defining the concept of a *control event*, which identifies the cross-products which are of interest. We then show how a graph of control events can be generated from a full state-graph and used for coverage analysis, with examples of practical application from the Stanford FLASH project [4], described in Section 3.

The drawback with this technique, as with all techniques that use state-space exploration, is the *state-explosion problem*. However, after analyzing the examples from FLASH, we have developed some techniques that allow us to obtain useful feedback despite large state-spaces. One technique performs an analysis of the RTL to find portions of the state-graph that can be pruned. This is possible in situations where the RTL is structured so that some of the state-variables can be *don't care* values. A second technique is to analyze an over-approximation of the state-space. These are both described in Section 4.

2 Control Events as a Coverage Metric

Ideally, we would like to make the identification of interesting behaviors in a design automatic. This is possible since the RTL-model of the design encapsulates the detailed description of its functionality. We can extract this information by translating the RTL to cooperating finite-state-machines (FSM) and finding the global state-graph.

* This work was supported by ARPA contract DABT63-94-C-0054 and Rockwell Semiconductor.

ICCAD '96
1063-6757/96 $5.00 © 1996 IEEE

This global state-graph contains *all* the possible behaviors of the design.

In practice, except for the simplest of designs, converting an entire circuit to FSM representation is infeasible. Instead, we focus on the control logic since many of the hard bugs are the result of multiple control-logic events, as shown in [5]. We can extract the FSMs associated with just the control sections and model the remaining parts of the design and the external environment *non-deterministically.* In the context of finding the global state-graph, non-determinism (ND) means trying all possible input sequences in all cycles. The ND-environment provides the most generality, it captures all possible behaviors of the FSM model. If a test-vector suite manages to exercise all interactions of the FSM model under the full ND-environment, we can assume good coverage of the interesting behaviors. Unfortunately, sometimes the ND-environment is too general and leads to control behaviors that are not possible in the design. When this situation arises, extra constraints need to be placed on the environment.

2.1 Defining Control Events

Since the *full* control state graph of a design is a comprehensive representation of its control behaviors, the straight-forward definition of coverage based on control *transition coverage* as defined in [3] provides a good starting point. This coverage metric presupposes that every transition of the control state graph must be tested to achieve full test coverage. There are two parts; a state coverage metric (SCM) and a transition coverage metric (TCM):

$$SCM = \frac{Number\ of\ States\ Visited}{Total\ Reachable\ States}$$

$$TCM = \frac{Number\ of\ Transitions\ Taken}{Total\ Reachable\ Transitions}$$

The SCM and TCM measures give numbers that represent how much of the total control state space has been tested. If both these metrics are 1, then we can be quite sure that every important test has been tried in the test-vector suite. However, our experience in using coverage metrics is that it is often difficult to reach an SCM or TCM of 1 except for the simplest of circuits. One reason is that in many designs, not all control transitions that appear in the full control state-graph need to be exercised to fully test functionality. A typical example is shown in Figure 2.1.

In this example, there are three variables, two of which control the datapath (*Var1* and *Var2*), with the third read only by the FSM representing *Var2*. If we take the full control state-graph, we have 8 states and 16 edges that need to be exercised in tests. However, some of these edges represent redundant tests. For example, the edge $2BX \rightarrow 3CX$ is equivalent to $2BX \rightarrow 3CY$ from a testing viewpoint since the datapath observes the same sets of commands from both edges. *Var3* can be ignored since its contribution to the behavior of the design has been made *explicit* by the state-space exploration that composed the individual FSMs. It does not directly control the datapath, so it is sufficient to exercise the edge $2B \rightarrow 3C$. This simply says that one or the other of those edges need to be tested, but not necessarily both. By applying this principle to the entire state-graph, we obtain the reduced graph shown in the lower right of Figure 2.1. This graph contains just 4 states and 5 edges that need to be exercised. We can generalize this observation by redefining the coverage metric in terms of possible *control events*.

Definition:

*A **control event** is a unique set of control variable values when projected onto the set of variables observed by the datapath.*

In other words, a control event represents a particular set of commands to the datapath. Intuitively, this gives a better measure of an interesting event than a simple cross-

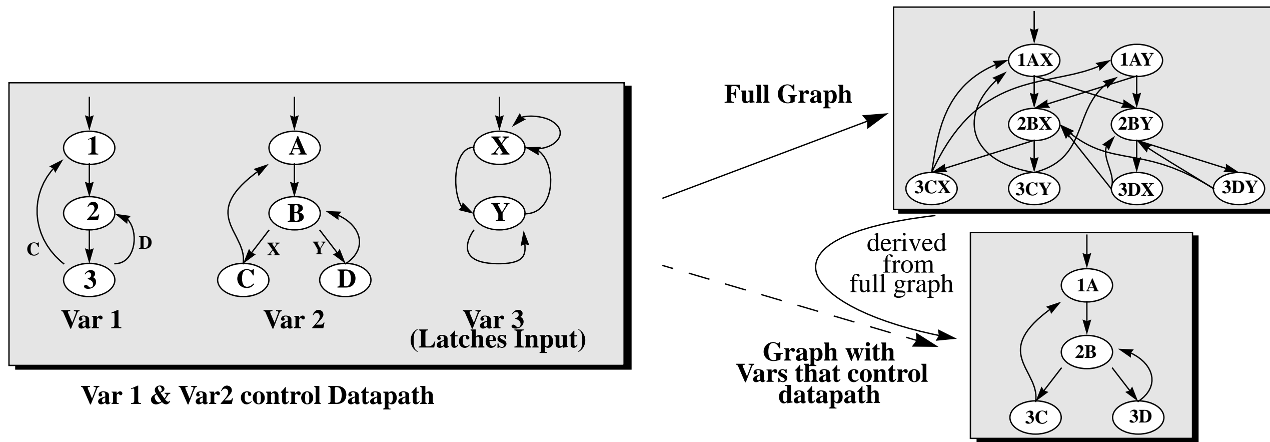

Figure 2.1. Example of State Graph Redundancy

product of all control-state variables since it takes into account which variables actually control the datapath actions and focuses our attention on the control-datapath interface.

2.2 Control Event Graph

A graph of control events can be created by projecting the full control state-graph onto the set of datapath-visible variables. A datapath-visible variable is one which directly, or through some combinational logic, controls the datapath. This graph represents all possible sequences of control events as given by the RTL description.

More formally, we can define a projection function *proj* (*proj:* $S \times \rho \rightarrow S_p$), where S is the finite set of states with each state composed of individual state variables $v_0 ... v_n$ from a set V of state variables; ρ is the finite set of sets of those state variables, ($\pi \in \rho, \pi \subseteq V$) and S_p is the set of finite states, each composed only of state variables in π, ($\pi \in \rho$). The function *proj* takes its two arguments, $s \in S$, $\pi \in \rho$, and returns a state s_p which is composed of only the state variables in π, such that if $v_i \in \pi$, then the value of v_i in s_p is equal to the value of v_i in s. Informally, *proj* extracts the state variables in the set π from s to create s_p. With the projection function, we can formally describe the control-event graph as a projection from the full control state-graph.

We can now cast the SCM and TCM metrics in terms of the control-event graph G_e to obtain a measure of coverage based on how the control logic interacts with the datapath:

$$SCM_e = \frac{Num.\ Control\ Events\ Visited}{Total\ Reachable\ Control\ Events}$$

$$TCM_e = \frac{Number\ of\ Transitions\ in\ Ge\ Taken}{Total\ Reachable\ Transitions\ in\ Ge}$$

The assumption made by control events is that the datapath does not hold any control state, so that only the sequencing of datapath commands matter, not their timing and duration.

3 Application to the Stanford FLASH Node Controller

This work used as a driving example, and was performed as part of, the Stanford FLASH (FLexible Architecture for SHared Memory) multiprocessor project. FLASH is a scalable shared-memory multiprocessor with up to 4k processing nodes. Each node contains a processor (MIPS R10000), a portion of the global memory and a flexible memory controller, called MAGIC (Memory And General Interconnect Controller). MAGIC contains an embedded RISC-processor core, some interfaces that manage and queue requests from several sources, an internal scheduler and a DRAM controller. A block diagram of the functional units of MAGIC is shown in Figure 3.1.

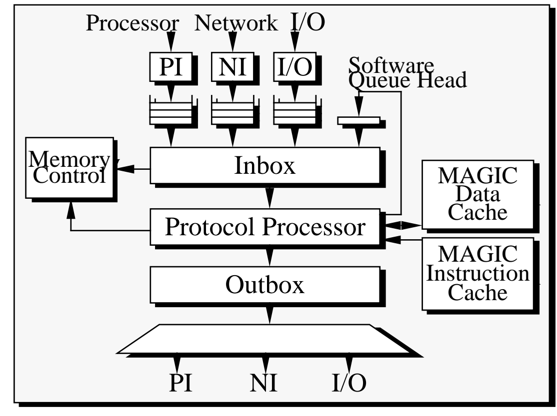

Figure 3.1. MAGIC Block Diagram

Self-checking diagnostics were used as the main validation technique. This was supplemented with some automatic test generation [5] for the Protocol Processor, for which an instruction set simulator provided the correctness check.

3.1 Coverage Analysis Toolset

To check the coverage of the validation tests, a coverage analysis toolset was implemented. This toolset consists of three steps: first, a translator that reads structural Verilog and produces an FSM description; second, a state-exploration program that creates a global state-graph; and finally, a coverage analysis program that implements control-event graph extraction and coverage marking. The three stages are illustrated in Figure 3.2.

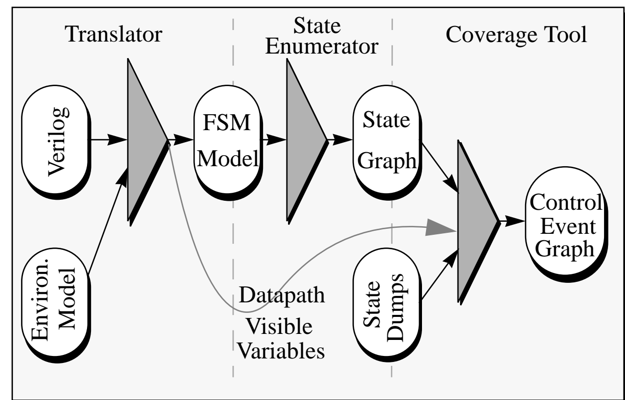

Figure 3.2. Coverage Analysis Toolset

The *translator* extracts the pertinent control-logic FSMs with the help of some user-supplied annotations in the Verilog. The annotations are comment-embedded directives that highlight some of the important state-variables in the control-logic. The translator then applies a *transitive set-of-support* algorithm to capture the logic which those state-variables depend on. The process stops

at the module boundary of the control-logic, based on the assumption that the design has already been partitioned into control and datapath sections for synthesis. The translator also performs the structural analysis of the Verilog description to find which variables directly, or through combinational logic, control the datapath and which are *independent*. This information is saved to file and used by the coverage program in the third step.

The FSM description language is a descendant of Murφ [6] called MPP (Murφ ++). The accompanying MPP verifier takes the FSM description of the system and finds all its reachable states from reset. It produces a global state-graph and hash-table of states.

The third step in the toolset takes the global state-graph and the set of independent variables to produce the control-event graph. It then reads state-dumps from tests run in simulation on the RTL. These state transitions are marked on the control-event graph and the individual state-variables, giving the coverage metric. Detailed feedback is given to the designer in the form of transition edges not exercised.

4 Coping with State Explosion

Any method that utilizes a state-space exploration needs to deal with the state-space explosion problem. Our experience has been that this can strike rapidly with even small changes in a design. We found that the design increased in complexity as it underwent timing tweaks: logic was moved around a fair amount to improve critical paths, some functions became pre-computed with a select, and some moved into other units. In general, interfaces became less clean and more state was introduced.

Although there is no universal solution to state-explosion, we have developed some techniques that help alleviate or postpone its onset so that some useful information can still be obtained even if the full state-space is too large to manipulate.

4.1 Graph Pruning using Don't Cares

In many large designs, it is often true that portions of the state-space are equivalent, meaning that pairs or groups of states can be represented by a single state without loss of information. Numerous techniques have been proposed to find such equivalences and hence reduce the state-space. Many of these have used the original graph as a starting point for finding equivalences, making them ineffective at dealing with the state-explosion problem. However, when dealing with state-graphs derived automatically from RTL descriptions, we have found that many equivalences can be traced back directly to the structure of the RTL. As a consequence of the logic structure, some sections of the resulting state-graph will inevitably be equivalent. For example, in Figure 4.1 we have a

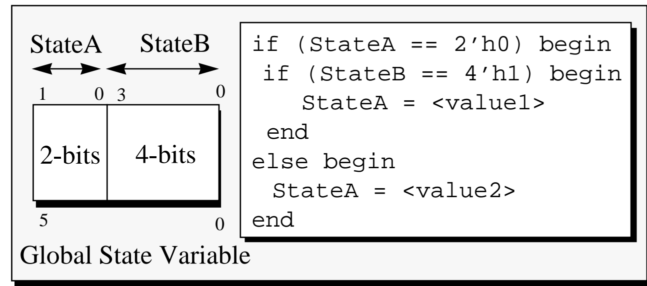

Figure 4.1. Don't Care Variable Analysis

global state variable that is composed of two component variables and we show a portion of the RTL code that sets the next-state. Assume *StateB* is set every cycle from an input. In this situation, the value of *StateB* is irrelevant when *StateA* is not equal to zero and can always safely be set to zero without losing any information. This is true since *StateB* is not even looked at when *StateA* is not zero and since *StateB* is set every cycle from the input, the next global state is the same no matter what value *StateB* is. So the sixteen possible values of *StateB* can be coalesced into a single representative state, reducing the overall state-space.

This is analogous to the kill-set in compilers, where a register is considered *dead* after it is last read and before it has a new value written to it. For state-space exploration, different values in that *dead* register show up redundantly as multiple states in the state-graph, whereas it is sufficient to just zero it without losing any information.

It would be possible to analyze the RTL structure very carefully to figure out dependencies and generate a complete kill-set for each variable. However, to be of real help in managing state-explosion, pruning the state-graph must occur dynamically and with minimum overhead. Hence, we introduce slightly stronger constraints on the kill-set to make recognition of pruning situations easy. Instead of doing multi-cycle analysis to figure out when a variable is written, we will impose the constraint that the variable to be pruned must be written every cycle. With this restriction, it is only necessary to determine if the variable to be pruned is read on any particular cycle. If not, it can be zeroed for that cycle.

4.1.1 Static Analysis of Kill-Sets

Analysis of the RTL for kill-sets occurs once the state-variables in the design have been determined. For each of the state-variables, we look for structures where the variable would not get read. A partial list of these structures is given in Figure 4.2. For every variable that is not read on occasion, we check the RTL to see if that variable gets a new value on every cycle. If so, we can construct a binary-decision-diagram (BDD) to represent the set of conditions where it can be ignored. The BDD simply

- Var. occurs in only one branch of an if-then-else statement.
- Var. occurs in a subset of all case branches.
- Var. is in a binary expression whose value is solely determined by the other variable, e.g. (0 AND x) or (1 OR x).

Figure 4.2. List of RTL Structures for Kill-Sets

encodes the values of the other state-variables that allow us to treat the variable being considered a *don't care* value.

4.1.2 Dynamic Pruning with Don't Cares

Once the kill-sets have been determined for every state-variable, we simply modify the state-enumeration tool to check each discovered state to see if it is one of the situations in which a state-variable is a *don't care*. If so, it zeroes out that variable before the hash table lookup, making all states that differ only in that state-variable equivalent.

Even with the more stringent conditions for kill-set creation that we imposed for efficiency reasons, we found that this dynamic pruning can provide some state savings with the examples from the FLASH project. Figure 4.3 shows the results for four units from MAGIC: the Inbox, Outbox, IO and PP. The PP was also split into two subparts: the instruction fetch unit and the loadstore unit. Each of these were exercised to greater or lesser extents by controlling the number of different input patterns applied. The results given show that for some models, such as the Inbox and the Outbox, dynamic pruning results in a substantial state-space reduction, accompanied by a reduction in the running time. For the other units, there was a small state-space reduction, but this was not enough to compensate for the runtime overhead of dynamic pruning. On closer examination of the kill-set BDDs from these units, it was obvious that the pruning potential of these models was small, that is, most state-variables did not fall into situations where they could be "killed". Hence, it was possible to determine after the static-analysis that these models were poor candidates for dynamic pruning, and it would be better to run the non-pruning version of state-exploration.

4.2 Approximating the State Space

Unfortunately, many interesting circuits have state-spaces that are too large to explore with available computing resources, even with pruning. In this situation, it would be beneficial to provide some useful information, even if it is not completely accurate.

One method is to approximate the model by treating some state variables as non-deterministic inputs. Instead of storing the value of these variables and using them to compute the next state, the transition function uses a non-deterministic value in their place. This reduces the size of the global state vector which, in general, reduces the size of the state-graph. Coverage can then be computed based on the reduced global state vector.

The problem is that constraints on the transition function from the exact state values are now lost. This introduces the possibility that some states found using the approximate state vector are actually unreachable in the real state-graph. Intuitively, this possibility is strongest when the state variable being replaced is part of the communication between other state variables retained in the global state vector. This implies that the best state variables to approximate are those that are close to the ND-inputs of the FSM model. Approximating these variables should have the smallest impact on the state graph.

To gain some empirical evidence for this, we approximated the state graph of the instruction fetch unit of MAGIC. The approximation was performed with two variables, *Var1*, which passes signals between state machines retained in the state vector, and *Var2*, which passes signals from the inputs. To measure the accuracy of the approximation, the exact graph was also found and then projected to the reduced global state vector for each variable. The results are shown in Table 1.

Table 1. Approximation Accuracy

	Approx. States	Actual States	Approx. Edges	Actual Edges	% Wrong
Var1	424	424	6,250	5,275	18.5%
Var2	322	322	4,943	4,887	1.2%

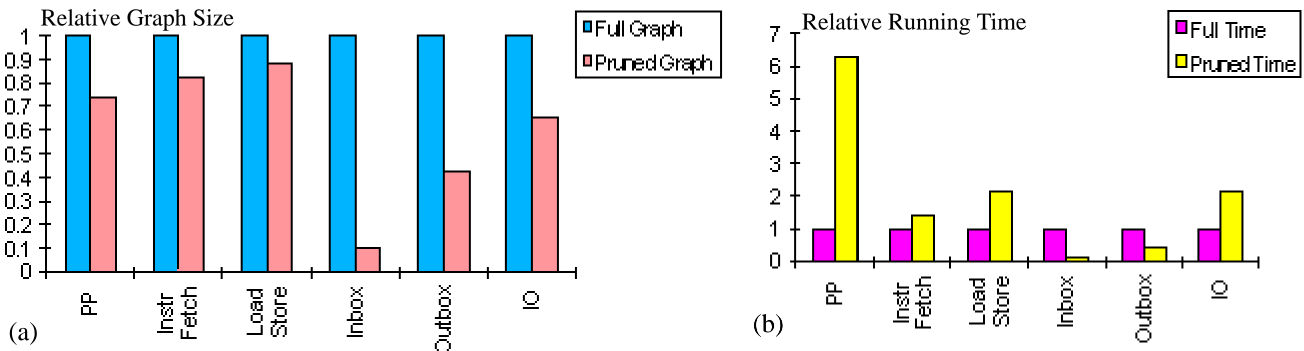

Figure 4.3. (a) Relative Graph Size and (b) Running Time of Pruned Graphs

Encouragingly, the results show that the approximation retained the correct number of states in both cases. However, the approximation with *Var1* resulted in a large number of false edges in the graph, while approximation with *Var2* was more accurate. This supports the heuristic of approximating state variables close to the inputs.

5 Coverage Results for FLASH

Coverage analysis of the FLASH design examples with the control-event metric, and pruning and approximation as needed, is given in Table 2. These results are a snapshot taken in the design process to show the difference between using a full state-graph metric and the control-event metric. The important point is *not* that the metric gives better numbers, but that it *highlights* the important tests which have been missed so that this information is not lost in a swamp of other redundant warnings.

The feedback from our design team was that incremental information is the most useful. When confronted with a coverage metric that indicates huge numbers of untested scenarios, it is difficult to identify the important cases missed. Instead, giving information about 1 FSM coverage, followed by pairs and so on, up to the full state vector gave the best results. When given only a few missing scenarios at a time, it was easier to identify new test vectors that had to be written to improve coverage.

In addition, it can be difficult to create a *particular* state transition of *all* state variables in simulation. Issues of controllability become important if the test writer must set up a state containing the full vector. By using control events and approximation, and doing so incrementally, the metric focuses attention on the variables which are important for the test cases only. The values of the other state variables, which do not play a part in the interaction, are ignored. This not only simplifies the analysis of the coverage data, but makes it easier for the test writer to formulate the new test that sets the variables to the needed values.

6 Conclusion

In this paper, we have proposed a new functional validation coverage analysis metric based on control events that focuses on the control-datapath interface. Our experience shows that this leads to better feedback of whether the important tests in a design have been exercised than existing metrics that we are aware of. The largest drawback is the use of the control state-graph, which potentially grows exponentially with the number of control state bits. Although this problem has no general solution, we have proposed two techniques that help to postpone the state-space blow-up so that useful information can still be obtained from the design.

This toolset was used in the process of designing a real chip as part of the Stanford FLASH project, making usability and graceful degradation important issues rather than afterthoughts.

References

[1] M. Kantrowitz, L. M. Noack, “Functional Verification of a Multiple-issue, Pipelined, Superscalar Alpha Processor - the Alpha 21164 CPU Chip”, In *Digital Technical Journal*, Vol. 7 No. 1 Fall 1995.

[2] A. K. Chandra, V. S. Iyengar, R. V. Jawalekar, et. al., “Architectural Verification of Processors Using Symbolic Instruction Graphs”, In *Proc. of the Intl. Conf. on Computer Design*, October 1994.

[3] Y. V. Hoskote, D. Moundanos, J. A. Abraham, “Automatic Extraction of the Control Flow Machine and Application to Evaluating Coverage of Verification Vectors”, In *Proc. of the Intl. Conf. on Computer Design*, October 1995.

[4] J. Kuskin, D. Ofelt, M. Heinrich, et. al., “The Stanford FLASH Multiprocessor”, In *Proceedings of the International Symposium on Computer Architecture*, June 1994.

[5] R. C. Ho, C. H. Yang, M. A. Horowitz, D. L. Dill, “Architectural Validation for Processors”, In *Proceedings of the International Symposium on Computer Architecture*, June 1995.

[6] D. L. Dill, A. J. Drexler, A. J. Hu, et. al., “Protocol Verification as a Hardware Design Aid”, In *Proceedings of the International Conference on Computer Design*, October 1992.

Table 2. FLASH Coverage Results

Unit of Design	Graph States	Graph Edges	Control Event States	Control Event Edges	Simulated Cycles	Control Event State Coverage	Control Event Edge Coverage	Full State Coverage	Full Edge Coverage
Proto. Proc.	22,080[a]	2,189,553	76[b]	184	794,342	30.3%	26.6%	<0.1%	<0.1%
Instr. Fetch	1,586[a]	14,455	17[b]	47	391,967	94.1%	59.6%	7.8%	2.0%
Load Store	12,192[a]	1,106,688	132[b]	532	685,683	28.8%	12.4%	<0.1%	<0.1%
Inbox	426	3,968	28	236	249,336	82.1%	28.0%	19.3%	3.0%
Outbox	52	506	14	62	101,756	92.9%	54.8%	53.6%	13.8%
IO	3,209	70,211	142	1,778	67,828	16.3%	6.7%	1.0%	<0.1%

a. Approximate state-space.

b. Designer restricted the variables of interest.