Toward Formalizing a Validation Methodology Using Simulation Coverage

Aarti Gupta
CCRL, NEC USA
Princeton, NJ

Sharad Malik
Princeton University
Princeton, NJ

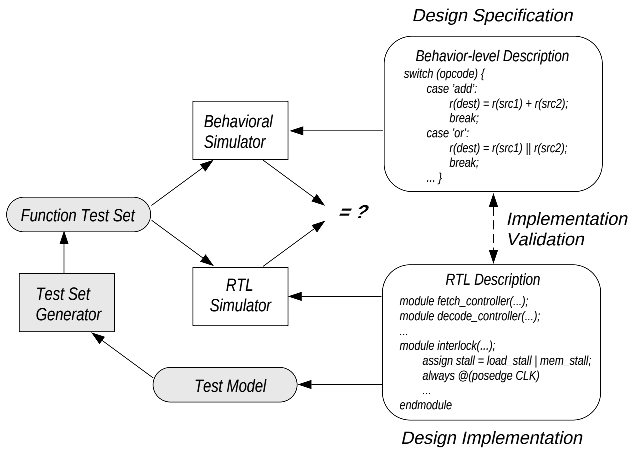
Pranav Ashar
CCRL, NEC USA
Princeton, NJ

ABSTRACT

The biggest obstacle in the formal verification of large designs is their very large state spaces, which cannot be handled even by techniques such as implicit state space traversal. The only viable solution in most cases is validation by functional simulation. Unfortunately, this has the drawbacks of high computational requirements due to the large number of test vectors needed, and the lack of adequate coverage measures to characterize the quality of a given test set. To overcome these limitations, there has been recent interest in hybrid techniques which combine the strengths of formal verification and simulation. Formal verification-based techniques are used on a test model (usually much smaller than the design) to derive a set of functional test vectors, which are then used for design validation through simulation. The test set generated typically satisfies some coverage measure on the test model. Recent research has proposed the use of state or transition coverage. However, no effort has been made to relate these measures to the coverage of design errors. Furthermore, the derivation of the test model remains largely ad-hoc, with few formal guidelines.
We demonstrate that under a given set of assumptions, transition tours on test models can be used for complete validation of an implementation against a specification, for a large and important class of designs that includes many programmable/hardwired, general-purpose processors/DSPs. A by-product of this study is specific guidelines for deriving the test model, motivated by the requirement of providing complete coverage of all errors. We illustrate the application of our methodology on a pipelined implementation of the DLX processor.

1 Introduction

An emerging paradigm in hardware validation is a hybrid methodology which combines functional simulation and formal verification. Formal verification-based techniques are used to derive a *test set* (consisting of *test vector sequences*), from a relatively simple *test model* of the design. This test set is then simulated on a *functional simulation model* of the design. The test set is selected such that it has certain coverage properties with respect to the test model – for example coverage of each state [18], or coverage of each transition [15]. While these techniques have been shown useful in locating design errors, it is not clear how these measures on the test model translate to coverage of design behaviors and errors, and guarantees with regard to completeness of validation. Furthermore, the derivation of the test model remains ad-hoc, with very few formal guidelines provided for this task.



```
switch (opcode) {
    case 'add':
        r(dest) = r(src1) + r(src2);
        break;
    case 'or':
        r(dest) = r(src1) || r(src2);
        break;
    ... }
```

```
module fetch_controller(...);
module decode_controller(...);
...
module interlock(...);
    assign stall = load_stall | mem_stall;
    always @(posedge CLK)
    ...
endmodule
```

Figure 1: Validation Methodology Using Simulation Coverage

This research attempts to fill these gaps by identifying classes of designs for which we can use coverage measures on test models to ensure complete coverage for design errors, under a reasonable set of assumptions. Specifically, we demonstrate how a *transition tour* (a test set that covers each transition) on a test model can be used for covering all errors with respect to a given specification, for many processors including programmable/hardwired, general-purpose processors/DSPs. This result is the first to characterize a subset of those designs which allow complete implementation validation through such a hybrid methodology. A by-product of this research is specific guidelines on deriving the test models for this subset. We believe this is an important step towards formalizing the use of validation techniques based on simulation coverage. On the practical side, we demonstrate the application of this methodology for the generation of test vectors for a moderately complex design – a pipelined DLX processor (with interlock detection, bypassing, squashing and stalling).

2 Overview: Simulation Coverage Methodology

Figure 1 shows the general framework of a simulation coverage-based methodology for implementation validation. In this section, we briefly describe the different components of this framework.
Since the design specification and the design implementation can be at different levels of abstraction (*e.g.* ISA description of a processor versus its RTL implementation), there may be no cycle-equivalent specification to compare the implementation against. The comparison between them is made at special checkpointing steps, *e.g.* at the completion of each instruction. To enable this, the implementation state used in this comparison is *observable* during functional simulation. For processors this tends to be most of the state of the datapath.
A test model can be derived either from the specification, or (as shown in Figure 1) from the implementation . It usually involves significant abstraction to reduce the state space complexity. For a processor, the test model typically retains only the control portion

of the design. The datapath is abstracted out since its state is observable during simulation. In a sense, the test model can be viewed as the non-observable part of the design. Since the datapath tends to be a large fraction of the total state, the test model is usually much simpler than the entire design.

A test generator then uses formal verification techniques to appropriately traverse (implicitly or explicitly) the reachable state space of the test model to provide the target coverage. In a transition (state) tour, each transition (state) is visited at least once. State space traversal and generation of transition/state tours tend to be computationally expensive even with implicit BDD-based methods [5, 25]. The generated test set is then applied to the functional simulation models of the specification and the implementation, with a comparison of outputs.

3 Related Work

The work described in this paper most closely relates to, and is indeed motivated by, the work reported in Ho *et al.* [15]. They use a transition tour on the implementation control FSM to automate test generation of corner cases, for validation of an embedded dual-issue pipelined processor. Their positive results highlight the practical impact of such a hybrid methodology. However, the issue of error coverage is not addressed. Our work attempts to formalize the requirements on the test model such that a transition tour can catch all transition/output errors with respect to the given specification. Much of the interest in transition tours has arisen from the area of conformance testing of protocols [10]. In particular, it is known that a transition tour can catch all errors if there exists an input which produces a unique output in each state, and causes the FSM to stay in the same state [10]. This property has a similar flavor to the property we use to motivate the completeness argument for our test model.

In general, function test vector generation/evaluation using formal models has been the subject of many efforts in specification validation. In contrast to implementation validation, there is no "golden" specification here, and the goal is to cover the design space for checking correctness. These techniques include – guaranteed coverage of every statement in an HDL description [8], evaluation of transition coverage of a given test set [16], abstraction of models and semantic control over transition coverage [13], and a coverage metric based on observability/error propagation [11]. Again, most of these coverage metrics do not provide a measure of the design error coverage.

In the more specific context of processor verification, the application of formal verification techniques has been limited by the high design complexity. Most automatic methods based on state-space exploration handle it either by considering smaller designs [2, 3], or by abstracting out the datapath to verify the pipelined control [6, 19, 20]. Formal verification attempts based on theorem-proving systems have also been successful [9, 17, 23], but require significant manual effort. At the simulation end of the spectrum, several efforts have focused on generation of effective function test vectors. The targets include architectural test sets [7], pipeline hazards [18], and property-specific architectural test sets [21].

4 Test Set Generation Using Transition Tours

In this section we examine properties of the test model that are needed for a transition tour to be sufficient for complete error coverage. The actual derivation of such a test model is described in Section 6.

4.1 Errors in Test Models

We consider the implementation to be a *Mealy machine*. The test model is derived from the implementation by abstracting its state

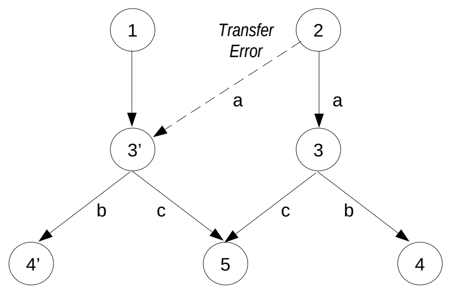


Figure 2: Limitations of Transition Tours: An Example

and input space. Since multiple transitions in the implementation, with possibly different outputs, may map to the same transition in the test model, the test model may have non-deterministic outputs. Let us now consider the possible errors in the implementation:

Definition 1 *A transition t is said to have an* output error *in the implementation if there is some sequence of inputs I that ends in t, for which its output value o is different from that of the specification.*

Definition 2 *A transition t is said to have a* uniform output error, *if its output in the implementation is different from that in the specification for all sequences of inputs I that end in t.*

Definition 3 *A transition t is said to have a* transfer error *if the destination state for that transition is different from what it should be in a correct implementation.*

A transfer error is considered to be *masked* if there is another transfer error for some subsequent transition that corrects the first one, i.e. control returns to the state which would have been reached had neither error been present. More formally:

Definition 4 *Let $< i_1, i_2, \ldots, i_n >$ be a sequence of inputs from some starting state. Let $< s_1, \ldots, s_{j-1}, s_j, \ldots, s_{l-1}, s_l, \ldots, s_n >$ be the sequence of states visited in a correct implementation, and let $< s_1, \ldots, s_{j-1}, s'_j, \ldots, s'_{l-1}, s_l, \ldots, s_n >$ be the sequence of states visited in an incorrect implementation. The transfer error from s_{j-1} to s'_j is said to be masked if the transition from s'_{l-1} to s_l is also a transfer error.*

Note that any error in the implementation is modeled as an output or a transfer error. This classification is motivated by the FSM fault model used in protocol conformance testing [10]. We now examine the coverage of these errors by transition tours.

4.2 Limitations of Transition Tours

Since a transition tour does not necessarily cover all sequences, it is easy to demonstrate its limitation in exposing all errors. Figure 2 shows a fragment of the state space of a test model. Consider the transition from State 2 to 3 on input a. Suppose that it is incorrectly implemented to go from State 2 to 3'. Assume that the transitions on input b, from State 3 to 4 and from State 3' to 4', are known to result in different outputs during simulation. Also, assume that the transition on input c, from State 3 to 5 and from State 3' to 5 result in the same outputs. Therefore, if a transition tour covers the transition on a from State 2 using the sequence $< a, c >$, as opposed to $< a, b >$, the transfer error will not be exposed. This illustrates the basic limitation of using transition tours – an error may be exposed only several transitions after it is excited only along a specific path in the state graph. If this path is not selected in the transition tour, it will not be exposed.

4.3 Complete Test Sets from Transition Tours

Let us now examine conditions under which a transition tour can generate a complete test set, i.e. a test set that uncovers all errors in the implementation. Since an error is associated with a transition by definition, it follows that a transition tour will excite all errors. The natural question that arises is: under what conditions can a transition tour also guarantee that all errors get exposed, i.e. each time an error is excited it is eventually exposed?
In this direction we now impose the following requirement:

Requirement 1 *All output errors are uniform.*

Intuitively, it hints at the right level of abstraction needed in the test model, i.e. it provides guidelines for how much of the implementation state should be present in the test model. We elaborate this point further in Section 6.
If Requirement 1 is satisfied, then for a transition which excites an output error in a test model, there is a corresponding transition in the implementation which exposes it. Thus, there is no dependence on a sequence of transitions to expose it. However, as illustrated in Figure 2, a transfer error may be dependent on a specific sequence of transitions for exposure. Therefore, to guarantee that it is exposed by a transition tour, we need to ensure that it is exposed by *all* sequences that follow it. More specifically, we need to ensure this for all sequences of length bounded by some k, since the simulator must also know how long to simulate in order to expose the error.
These observations motivate the following definition and theorem:

Definition 5 *A state s_1 in the test model is said to be $\forall k$-distinguishable from a state s_2, if all input sequences of length k can distinguish them.*

Theorem 1 *If Requirement 1 is satisfied, and if all states in the test model are $\forall k$-distinguishable from each other for some fixed k, then a transition tour of the test model is sufficient to expose all errors through simulation.*

Proof: The proof follows directly from the fact that an output error on a transition must be exposed by that transition itself (since all output errors are uniform), and a transfer error is exposed by any sequence of length k that follows it in the tour. ■

While *uniformity* of output errors and $\forall k$*-distinguishability* of states are strong properties, they give us precisely the class of test models for which a transition tour provides complete error coverage.
There is a subtle point that needs further elaboration. A test sequence for the test model needs to be converted to a test sequence for the implementation simulation model since some of the inputs may have been abstracted out in the derivation of the test model. While Theorem 1 guarantees the existence of complete test sequences for the simulation model for the implementation, it does not point out how they are derived from the test sequence for the test model. This involves a careful selection of the inputs being abstracted and is beyond the scope of current discussion.

5 Verification for Processors

We now examine the properties presented in Section 4 for a large class of practically useful designs that fall under the general category of processors. In the case of a programmable processor (e.g. general purpose processor/digital signal processor) the input sequence may be a sequence of instructions and data values. In the case of a fixed program processor (e.g. a signal processing ASIC) the input sequence is simply a sequence of data values. After processing each input, it outputs the result in some observable form. These outputs may be values on ports of the design, or observable data state in registers and/or other memory.
Both the processing function and the output actions are typically specified as part of an architectural specification. What primarily distinguishes this class of designs from other finite state machines is that a large part of the implementation state is observable as outputs, which can be compared with the architectural specification during functional simulation. It is precisely this feature that makes them attractive for simulation-based verification. Furthermore, the large observable data state is effectively used for comparison during functional simulation, but plays little role in flow of control, and thus need not be considered in deriving the test model.
On the other hand, the differences in the levels of the specification and the implementation permit a wide range of possible implementations. For example, an implementation may introduce parallelism in the processing of instructions in the form of pipelined or superscalar execution. Thus, there may be multiple instructions in different stages of execution at the same time. This parallelism makes the validation task for such designs difficult.
Consider the processing of a sequence of inputs $< i_0, i_1, \ldots, i_n >$. Due to the available parallelism, several inputs may be processed between the time when processing starts for input i_j and when its output is observable. We impose the following requirements on test models of processors being considered.

Requirement 2 *The processing required to generate the output for each input completes in at most k transitions.*

Requirement 3 *Each unique input results in a unique output.*

The bounded execution time imposed by Requirement 2 seems reasonable for most contemporary processors. Requirement 3 can be satisfied by appropriately picking data values that distinguish the outputs for various instructions.

5.1 $\forall k$-Distinguishability for Processors

We now examine the $\forall k$-distinguishability property for test models for processor designs.
We regard the state s of the test model as being composed of two parts: $s = s^1 \times s^2$. The first part, s^1, captures the state needed for computing the outputs for all inputs being currently processed. The second part, s^2, captures the state needed by subsequent inputs, i.e the potential interactions with inputs yet to be processed. (If some part of the state qualifies for being in both s^1 and s^2, it is considered as part of s^2.) For example, in a pipelined processor, s^1 corresponds to all instructions currently in the pipe, while s^2 may include (among other things) the `zero` flag of the Processor Status Word, which might be needed by a subsequent `beqz` (branch when equal to zero) instruction. Some interactions arise due to simultaneous processing of instructions also. For example, in a pipelined processor, two successive instructions can interact if the destination register of the earlier instruction is the same as a source register for the following one. Therefore the destination register address will also be part of s^2.
We now use this partitioning of the state to examine the $\forall k$-distinguishability property for each pair of distinct states s_m and s_n. Recall that this property was needed to expose a transfer error in a transition on some input i_q, where the transition goes to state s_n instead of state s_m. There are two possibilities:

Case 1: $s_m^1 \neq s_n^1$
We impose the following requirement on transfer errors:

Requirement 4 *Transfer errors are not masked.*

This requirement makes sure that the transfer error on i_q does not get rectified by subsequent inputs. Therefore, if the s^1 part is corrupted, it will show up in the observed outputs (by definition of the s^1 part). From Requirement 2, the processing of input i_q can take no more than k transitions through the test model. Also, from Requirement 3 each unique input will result in a unique output. This implies that at the end of all sequences of length k, the outputs will be different starting from these two states, i.e. s_m and s_n satisfy the $\forall k$-distinguishability property.

Case 2: $s_m^1 = s_n^1$, $s_m^2 \neq s_n^2$
Since that part of the state which determines outputs is the same, there is no guarantee that all following sequences of inputs will expose the transfer error. In fact, the error may get exposed only for a very specific input sequence. For example, consider a pipelined processor where the destination register address is part of the s^2 state. Let instruction i_q use R_0 as its destination register. Suppose the transfer error causes the transition on input i_q to go to a state with s^2 component corresponding to R_1 instead of R_0. The only way to expose this error is to have the subsequent instruction use R_0 as a source register. However, this sequence may not be selected by the transition tour, thereby leaving the error un-exposed.
One solution for this case is to make the s_2 component observable, i.e. it is observable as an output of the functional simulation model. With this solution, s_m and s_n are now directly distinguishable since the two states have different outputs. Thus they satisfy the $\forall k$-distinguishability property. We impose this solution as a requirement on the design.

Requirement 5 *The state associated with interactions between processing of subsequent inputs is made observable.*

The set of prescribed requirements now enables us the state the following theorems.

Theorem 2 *If Requirements 2, 3, 4 and 5 are satisfied, then the test model satisfies the $\forall k$-distinguishability property, i.e. any state can be distinguished from any other state for all sequences of length k.*

Proof: Follows from the discussion of the two cases. ■

Theorem 3 *If Requirements 1, 2, 3, 4 and 5 are satisfied, a transition tour of the test model provides a complete test set.*

Proof: Follows directly from Theorems 1 and 2. ■

Note that these results depend on the specific requirements being satisfied. The onus of satisfying them rests on the test model. In the next section we describe how a test model is derived from an implementation, keeping these requirements in mind.

6 Guidelines for Derivation of the Test Model

6.1 Test Models as Implementation Abstractions

As pointed out earlier, our test model is an abstraction of the design implementation. We use abstraction to remove those parts of state which are directly observable for comparison against the specification, and which do not influence the control flow. Consider the example of a pipelined processor which has five pipeline stages. The entire state in its implementation can be captured as:

$$S_c = \{i_1 \times i_2 \times i_3 \times i_4 \times i_5 \times p_1 \times p_2 \times p_3 \times p_4 \times p_5 \times F \times R\} \quad (1)$$

Here i_j is the instruction in stage j of the pipeline, p_j is the additional state associated with stage j, F is the state associated with various flags in the processor, and R is the state associated with the registers. (For exposition purposes, we do not consider other memory in the processor.) We refer to this as the *concrete* (non-abstract) state space. We can abstract out components that do not affect the control flow, e.g. contents of registers. Moreover, the entire instruction may not be needed by each stage of the pipeline, and we let the relevant parts be included within each pipeline stage state p'_j. Thus, it is possible to use the following abstract state space:

$$S_a = \{p'_1 \times p'_2 \times p'_3 \times p'_4 \times p'_5 \times F\} \quad (2)$$

In general, we use a homomorphic abstraction which is a many-to-one mapping $\mathcal{A}$ from states in the set S_c (concrete states) to states in the set S_a (abstract states). Furthermore, this mapping preserves the transition relation, such that a transition t_c between states s_1 and s_2 in the concrete state space maps to a transition t_a between $\mathcal{A}(s_1)$ and $\mathcal{A}(s_2)$ in the abstract state space. While $\mathcal{A}$ provides for a general mapping over *states*, in practice, abstractions tend to have the special form of a mapping over *state variables*. Note that such an abstraction needs to examine state variables only, and not the entire set of states. This logarithmic reduction in complexity is the reason why this special form of abstraction is attractive and almost universally used.
In practice, an abstraction over state variables can be implemented by removing certain state elements from the concrete model, and all of the logic associated with only that part – this is a simple topological operation. Any communication signals between the abstract model and the parts abstracted out are now considered as input/output signals for the abstract model. The outputs are not a problem, since they are not needed during test set generation, but the inputs need special attention. For example, consider the Processor Status Word of a processor. If the datapath is abstracted out, these signals are modeled as independent inputs to the test model. However, during functional simulation, they are generated by the datapath and are no longer available as independent input signals. We propose using the solution suggested by Ho *et al.*, which is to take control of these signals during functional simulation also, thus permitting the direct application of the generated test set.

6.2 Ensuring the $\forall k$-Distinguishability Property

The state-merging and transition-preserving nature of the many-to-one mapping $\mathcal{A}$ has an important consequence. If the concrete test model has the $\forall k$-distinguishability property, then so does the abstract test model. Formally, let s_a and $s_a{}'$ be two distinct states in the abstract model, where $s_a = \mathcal{A}(s_c)$ and $s_a{}' = \mathcal{A}(s_c{}')$. Since s_c and $s_c{}'$ are distinguishable for all sequences of length k, due to the transition preserving nature of $\mathcal{A}$, these sequences will also distinguish between s_a and $s_a{}'$.
Therefore, once this property is ensured for a test model, all subsequent abstractions inherit it. In Section 5, Theorem 2 described the requirements needed to ensure this property. In the remainder of this section we discuss practical ways of enforcing these requirements. To start with, we need to assume that Requirements 2 and 4 hold naturally. Next, Requirement 3 does not need special consideration during derivation of the test model. Given a test model, it can be enforced simply by picking appropriate data during simulation such that each instruction gives a unique output. Requirement 5 involved making observable the state corresponding to interactions between inputs. It is useful to examine the practical issues here. In general, capturing the state corresponding to input interactions requires an understanding of the design. In fact, such information is typically known to the designers, though not formally stated. Furthermore, the designer (or the test model developer) does not need to examine the entire state space of interactions. Rather, they just need to identify the state variables involved. This is linear in the size of the total state vector. Therefore, we believe it is manageable in practice, and are currently working on formalizing it in an effort towards automation. For example, in the case of the DLX processor described in Section 7, the only state elements needed for this purpose are the flags in the Processor Status Word and the address of the destination register for register instructions. Note that the exact nature of the hazard resolution is not relevant, just the fact that the address of the destination register represents the state useful for interaction with subsequent instructions.

6.3 Abstracting Too Much

Thus far the only restriction on the abstraction mapping is that it be transition-preserving. If there were no restrictions on how much we could abstract, we could abstract down to a single state! Obviously this would not provide enough detail for the test model. Thus, we need some measure on how much we can abstract without losing critical information. Requirement 1 provides us with precisely this measure. It states that for an output error on transition t, the error must be exposed by all sequences that end in t, i.e. the detection of this error must not be a function of any preceding sequence. If it is indeed a function of the preceding sequence, then this is an indication that the test model has not saved enough state to distinguish between those preceding sequences that expose the output error and those that do not.

As an example, consider the read-after-write interlock in a pipelined processor. If the state in the test model does not store the address of the destination register, then a transition will not expose an error in the interlock mechanism for all preceding sequences. It will do so only when the preceding instruction uses the same destination register, say R_0, as the current source register. Thus, the output error on t is not uniform, in violation of Requirement 1. On the other hand, if the destination register is made part of the test model state, any transition from such a state on an instruction which uses R_0 will expose an interlock error (if it exists), regardless of what preceded it. Thus an output error on this transition is uniform.

We have used the interlock mechanism example to illustrate two major issues in derivation of test models, and it is important to clarify the distinction between them. In Sections 5.1 and 6.2 we stated that the address of the destination register must be made visible. This was done to ensure the $\forall k$-distinguishability property of test model states. In this section, we have emphasized the need for including it in the test model state to ensure that there will exist a transition (and not a sequence of transitions) in the test model that will detect the output error, i.e. to ensure the uniformity of the output error as stated in Requirement 1.

6.4 Impact of Incorrect Implementations

We now examine the issue of how an incorrect implementation might affect the previously stated requirements. As mentioned earlier, Requirements 2 and 4 are regarded as assumptions. Also, Requirement 3 is handled by picking appropriate data values during simulation, and as such is unaffected by an incorrect implementation. Thus, only Requirements 1 and 5 may get affected when a test model is derived from an incorrect implementation. As discussed in detail, Requirement 1 deals with abstractions, and Requirement 5 deals with exposing interaction state to make it observable. Thus, the only requirement on the implementation is that it have all the state of a “correct" implementation. There is no requirement on the correctness of the transitions, in fact, a transition tour will expose both the output and the transfer errors on transitions. Ensuring that the implementation has the correct states may not be a significant burden – typically errors creep in on the transitions.

6.5 Generating Test Set from the Test Model

Once a test model is available, the remaining task is to perform a transition tour on its state space to generate the test set. It is known that the problem of finding a minimum cost transition tour corresponds directly to the *Chinese postman problem*, which can be solved in polynomial time [1]. For our implementation, we use a BDD-based transition relation representation of the test model within the SIS system [22]. A transition tour is generated by traversal of this implicit representation, along with consideration of input don't-cares. Since the inputs to the test model are abstracted from those for the actual design, appropriate input values must be filled in before the generated test set can be used for simulation.

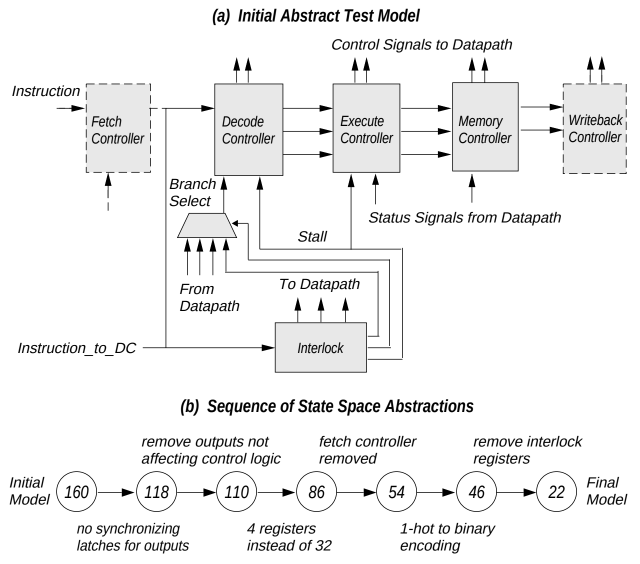


Figure 3: Abstract Test Model for a Pipelined DLX Implementation

7 Case Study: The DLX Processor

The DLX architecture was introduced by Hennessy and Patterson [14] to illustrate the basic RISC concepts. Though considerably less complex than contemporary RISC processors, it incorporates many advanced features that make it an interesting case study of medium complexity. In this section, we report on our experiences with deriving a test set for an implementation of this processor.

We used an RTL implementation in Verilog [24], provided to us by Prof. Franzon at NSCU [12]. Undertaken as a class project, this design implements the DLX instruction set (except the floating-point and exception-handling instructions). It uses a standard 5-stage pipeline and includes an interlock module for handling various pipeline hazards.

7.1 Deriving the Test Model

We started by abstracting out all the datapath modules. This resulted in the model shown in Figure 3 (a), consisting of individual controllers for the 5 stages of the pipeline, the interlock unit and the multiplexor used for selecting the branch test result. Note that in this model, the signals from/to the datapath (including the instruction word) are modeled as primary inputs/outputs, respectively. As discussed in detail in Section 5, our test model needs to capture interaction between different instructions, other than those through observable data itself. For the DLX architecture, such interactions are captured by:

- addresses of destination registers from the current, and two previous, instructions
- the Processor Status Word

The destination addresses are already a part of the initial model, and we only need to be careful not to abstract them out. As for the Processor Status Word, these signals are modeled as primary inputs to the test model since they arise from the datapath which has been abstracted out. This suffices because test vectors are generated for all possible values of the primary inputs. Finally, since the test model does not require any state corresponding to actual data, it is possible to remove all signals between controllers used for transferring immediate data.

For our design, this initial model contained 160 state elements (register variables), 41 primary inputs, and 32 primary outputs.

Since this was most likely beyond the capabilities of current state-based tools, we focused on using abstractions that would preserve completeness of our test model, while reducing the number of state elements as well as primary inputs.

- **Abstraction over State Space**: The sequence of our state space abstractions (with number of state elements) is summarized in Figure 3 (b). Note that the first four abstractions we used are quite general, and they can likely be applied to any pipelined design. The last two are quite specific to the design we worked with, and may not be even required for an implementation style that is more efficient to start with. In both cases we use local transformations that we assume are correct (or can be easily proved) and make no assumption about the overall function of the design. The final test model had 22 latches.
- **Abstraction over Primary Inputs**: For the standard DLX architecture, the instruction format consists of 32 bits. However, the instruction input for our test model differs from the standard instruction in the following ways – all immediate data fields can be removed, and only 2-bit address fields are required for 4 registers in the register file. This resulted in an 18-bit instruction format for the test model. Note that this reduced format still captures all the different instructions in the design.

7.2 Experimental Results

Our final model in Verilog contains 22 latches, 25 primary inputs and 4 primary outputs. We used VIS [4] to convert the Verilog description to an FSM description, which was further used as input to SIS [22]. Within SIS, the implicit transition relation representation of the model was obtained in about 10 seconds on an Ultrasparc (166 MHz.) workstation with 64Mb. main memory. Though there are 25 primary inputs in the model, not all combinations are allowed due to invalid instructions and relationships between datapath outputs modeled as primary inputs. Of the 2^{25} possible input combinations, only 8228 are valid combinations. Taking input don't-cares into account reduces the number of reachable states as well as the number of transitions that need to be visited.
The model has 13,720 reachable states, much less than the possible 2^{22}. This observation is similar to that made by Ho *et al.* in their experiments [15]. We determined the model to have 123 million transitions, and a tour of length 1069 million transitions. This is not an optimal tour, and we are currently working on generation of more efficient tours.

8 Summary

This research was prompted by recent efforts in using hybrid techniques combining simulation and formal verification for design validation. These efforts leave unresolved two main issues – completeness of the test vector set, and guidelines for generating the test model. This research examines these issues and highlights their relationship. Specifically, we show that under a set of reasonable assumptions, a transition tour of a test model provides a complete test set for a large subset of designs classified as processors. We describe the requirements this imposes on the test model, and provide specific guidelines for satisfying them.

REFERENCES

[1] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. *IEEE Tran. on Comm.*, 39(11):1604–1615, Nov. 1991.

[2] D. L. Beatty and R. E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Proc. 31st ACM/IEEE Design Automation Conf.*, pages 596–602. IEEE Comp. Soc. Press, June 1994.

[3] V. Bhagwati and S. Devadas. Automatic verification of pipelined microprocessors. In *Proc. 31st ACM/IEEE Design Automation Conf.*, pages 603–608, June 1994.

[4] R. K. Brayton et al. VIS: A system for verification and synthesis. In *Proc. Int. Conf. on Comput.-Aided Verification*, volume 1102 of *LNCS*, pages 428–432. Springer-Verlag, July 1996.

[5] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Tran. on Computers*, C-35(8):677–691, Aug. 1986.

[6] J. R. Burch. Techniques for verifying superscalar microprocessors. In *Proc. 33rd ACM/IEEE Design Automation Conf.*, June 1996.

[7] A. K. Chandra et al. Avpgen – a test case generator for architecture verification. *IEEE Transactions on VLSI Systems*, 6(6), June 1995.

[8] K. Cheng and A. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proc. 30th ACM/IEEE Design Automation Conf.*, pages 86–91, June 1993.

[9] A. Cohn. A proof of correctness of the VIPER microprocessor: The first level. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71. Kluwer Academic Publishers, 1987.

[10] A. T. Dahbura, K. K. Sabnani, and M. U. Uyar. Formal methods for generating protocol conformance test sequences. *Proc. IEEE*, 78(8):1317–1326, Aug. 1990.

[11] S. Devadas, A. Ghosh, and K. Keutzer. An observability-based code coverage metric for functional simulation. In *Proc. IEEE Int. Conf. on Comput.-Aided Design*, pages 418–425, Nov. 1996.

[12] P. Franzon. Digital computer technology and design: Fall 1994 project. Private Communication, 1996.

[13] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolsfthal. Coverage-directed test generation using symbolic techniques. In *Proc. Int. Conf. on Formal Methods in CAD*, Nov. 1996.

[14] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[15] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill. Architecture validation for processors. In *Proc. 22nd Annual International Symposium on Computer Architecture*, June 1995.

[16] Y. Hoskote, D. Moundanos, and J. A. Abraham. Automatic extraction of the control flow machine and application to evaluating coverage of verification vectors. In *Proc. IEEE Int. Conf. on Comput. Design*, pages 532–537, Oct. 1995.

[17] W. A. Hunt, Jr. Microprocessor design verification. *J. Automated Reasoning*, 5(4):429–460, 1989.

[18] H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose. Automatic test program generation for pipelined processors. In *Proc. IEEE Int. Conf. on Comput. Design*, pages 580–583, Oct. 1994.

[19] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *Proc. IEEE Int. Conf. on Comput.-Aided Design*, pages 2–6, Nov. 1995.

[20] J. Levitt and K. Olukotun. A scalable formal verification methodology for pipelined microprocessors. In *Proc. 33rd ACM/IEEE Design Automation Conf.*, pages 558–563, June 1996.

[21] D. Lewin, D. Lorenz, and S. Ur. A methdology for processor implementation verification. In *Proc. Int. Conf. on Formal Methods in CAD*, Nov. 1996.

[22] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proc. IEEE Int. Conf. on Comput. Design*, 1992.

[23] M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, Sep. 1990.

[24] D. E. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.

[25] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *Proc. IEEE Int. Conf. on Comput.-Aided Design*, pages 130–133. IEEE Comp. Soc. Press, 1990.