Effectiveness of Microarchitecture Test Program Generation

Noppanunt Utamaphethai
R.D. Shawn Blanton
John Paul Shen
Carnegie Mellon University

Investigation of a list of design errors typically encountered in industry is undertaken to determine if our microarchitecture test programs can detect them. Two metrics, functional and timing deviation are used to determine design error coverage.

Microprocessor performance has doubled every eighteen months for the past two decades. Improvements in the semiconductor manufacturing process, the use of deeper pipelines, and aggressive microarchitecture mechanisms are among the important factors in the impressive increase. However, higher performance is accompanied by design complexity because of multifaceted microarchitecture mechanisms for extracting instruction-level parallelism. Dominant mechanisms include dynamic branch prediction to ease the constraints imposed by control flow dependencies; register renaming to remove the unnecessary serialization imposed by false data dependencies; reservation stations to buffer instructions awaiting their operands without having to stall the fetch and decode stages; out-of-order issuing to avoid the unnecessary stalling of subsequent ready instructions; and reorder buffer to support inorder completion and precise exception even when instructions are executed out of order.

Design validation for the state-of-the-art microprocessors has become a more difficult and time consuming task because of the increased design complexity. Validation has become important enough that many companies have adopted a “2-for-1” strategy; two personnel are used to check the work of every one designer. Simulation is the current industry practice for validating these aggressive microarchitecture mechanisms. Simulation models are developed early, then maintained and used throughout the design cycle. Design validation involves exercising the models and examining the outcome. To exercise these models for validation, instructions or real register transfer-level (RTL) signal sequences are given as input stimuli. Real application programs, hand-generated sequences, and randomly generated sequences are frequently used. While real programs may represent typical user workload, they may not fully exercise the design. Randomly generated sequences can exercise the design more completely, but are inefficient because of the large number of simulation cycles required. Even though explicitly generated test sequences can be efficient in terms of the number of simulation cycles, they are usually produced in an ad hoc fashion based on the intuitive knowledge of the designer. Hence, there is a real need for a systematic method for generating efficient design validation test sequences based on rigorous and
efficient representations of the microarchitecture mechanisms.

Formal verification is a viable approach to ensure design correctness.\textsuperscript{1–5} It has proven quite successful in verifying logic circuits. More recently, these formal techniques were extended to verifying microprocessors. However, only a portion of a complex microprocessor or a relatively simple microprocessor can be verified with these formal methods.\textsuperscript{6,7}

A key problem with formal verification methods is complexity explosion. Although the efficiency of formal verification algorithms has been significantly improving, the complexity of contemporary superscalar designs appears to increase at an even faster rate. This complexity problem is exacerbated by two dominant trends in contemporary superscalar design that antagonize the current strengths of formal methods. First, formal verification techniques have been successfully applied to the data path portion of a microprocessor, and shown to be quite effective.\textsuperscript{7,8} However, the control logic for managing simultaneous, in-flight instructions in many pipe stages is what makes the validation of modern superscalar machines difficult.

Recent research efforts regarding formal methods have focused on verifying pipelined designs.\textsuperscript{6,7} These techniques are most effective when the pipeline is fairly shallow and straightforward, with a single sequential path for instruction flow and minimal interaction between different pipe stages. Modern superscalar design trends are contrary to these assumptions. They are becoming deeper in terms of pipe depth with increasing interactions between different pipe stages. For example, branch predictors are accessed during the fetch stage but must be updated by the branch execution unit in the execute stage. Register renaming is performed in the dispatch stage, but rename registers are retired by the write-back stage. In addition to advancing instructions to the reservation stations in the next pipe stage, the dispatch stage must also allocate entries in the reorder buffer, which is usually viewed as part of the completion stage. The validation of these complex interactions between pipe stages is crucial and poses the most difficult validation and verification problem.

In addressing the complexity problem, a number of recently published papers attempt to leverage specific algorithms from formal verification but adopt a more simulation-based and/or testing-oriented methodology. One emerging paradigm combines formal verification and functional-level simulation. Most work derives finite-state machine (FSM) models of the design from the structural implementation or uses the ones provided by the designer.\textsuperscript{10–13} The resulting model is likely to be limited to the functionality of a single pipe stage. As pointed out earlier, microarchitecture mechanisms generally involve interactions between many pipe stages. Consequently, representing a microarchitecture mechanism requires the composition of many FSMs, which then leads to the aforementioned complexity explosion. To alleviate the complexity explosion problem, some abstraction techniques are applied to the finite state machines so that formal verification techniques, such as property checking during state enumeration and transition traversal, can be used to generate tests.\textsuperscript{10,13,14} In general, a set of tests is generated to maximize some coverage metric (such as transition coverage).

Buffer-Oriented Microarchitecture Validation

In our previous work, we introduced the buffer-oriented microarchitecture validation (BMV) method, which operates at the microarchitecture level of abstraction and targets the validation of critical microarchitecture mechanisms.\textsuperscript{15–17} While pipeline stages and major functional units are explicitly represented at this level of abstraction, the focus is more on validating the behavior of the major microarchitecture mechanisms and not necessarily the actual RTL implementation of these mechanisms. The goal is to ensure that the microarchitecture mechanisms, such as branch prediction, register renaming, and out-of-order execution, function as the microarchitect intends. The BMV method targets the control aspect of the microarchitecture that involves interactions between instructions in different machine pipeline stages, and nicely complements formal verification techniques for data path functional units.
Validation of the entire machine is impractical as a single unit. To alleviate the complexity problem, the method of partitioning the entire machine into multiple modules for separate validation is always used. We propose a different approach to machine partitioning than that based on pipeline stages. Examination of contemporary processors reveals that many operations are associated with microarchitecture memory buffers. For example, branch prediction involves the reading of the branch target address cache in the fetch stage and the branch history table in the decode/dispatch stage. During branch resolution, the branch execution unit updates these buffers. Partitioning within the BMV method is based on these and other buffers.

Each microarchitecture memory buffer consists of identical entries that range from a few to several dozen, and potentially several hundred in future designs. Each buffer entry stores data (operands or an instruction) and possibly status bits that indicate the status or the state of the data being stored. These status bits are usually used as control signals to trigger subsequent events and carry out the control function of the machine. Entry operation involves reading, writing, and updating the data and/or the status bits of the entry.

The key concept of the BMV method is to view microarchitecture mechanisms as specific operations on the data or the status bits stored in the entries of the microarchitecture memory buffers in the processor. The control aspect of the microarchitecture can be viewed as the management of the operations performed on buffer entries. Figure 1 shows the buffers (as shaded boxes) involved in the control aspect of the microarchitecture of the PowerPC 604.
PowerPC 604. The stored status bits can be viewed as the state bits of an entry. When the status bits are updated, the update process can be viewed as the entry transitioning from one state to another. Consequently, each buffer entry can be modeled as a small FSM.

Since all buffer entries are identical, the entire buffer behavior can be characterized by an entry behavior and how it is accessed. In traditional logic testing, an iterative array structured circuit can be elegantly tested by partitioning the circuit into its identical modules, each of which is tested identically. We leverage this concept in our BMV method. Since the buffer entries are identical, a buffer is validated by identically validating each of its entries. Each entry can be validated by exercising its FSM model. Specific FSM testing techniques that use a state tour, a transition tour, or a checking sequence can be used to validate the buffer’s functionality. Most recently, there has been a path-based (such as a sequence of FSM transitions) metric described that is effective in finding bugs. The FSM test technique employed affects how thoroughly exercised the model is and the effort required to generate and apply such a technique.

The methodology in the BMV is to determine the buffer type functionality, model its operations at the microarchitecture level using abstract FSM models, and rigorously generate test programs that systematically exercise all FSM transitions. We call this last step microarchitecture test program generation (MTPG). By (manually) extracting the control functionality of an entry in a given buffer and representing it using a FSM model, the microarchitecture abstraction is achieved.

Each test program is generated by first deriving a high-level test specified as a sequence of FSM transitions. We choose a complete transition tour as a high-level test as it provides a good trade-off between sequence generation effort and the validation level achieved. Small PowerPC assembly instruction sequences are derived and used to translate the FSM transition tours into a simulatable instruction sequence, such as an assembly test program. These small PowerPC assembly instruction sequences are called atomic sequences. Since each atomic sequence is associated with a transition in the FSM model, the high-level tests are translated by concatenating the atomic sequences in the same order as the FSM transitions appear in the transition tour.

Real and randomly generated programs seldom serve as efficient sequences for exercising FSM buffer models. Even if all FSM transitions are traversed using a randomly generated program, the number of random instructions is far greater than our systematically derived test programs based directly on the FSMs. While this is encouraging, the true effectiveness of our test programs can be judged only in terms of how well they uncover real design errors.

Contemporary Microarchitecture Mechanisms

The PowerPC 604 is a four-wide superscalar microprocessor with branch prediction and out-of-order execution. Branch prediction attempts to determine the instruction stream flow so instructions and operands are prefetched in advance. Independent instructions may be executed in an order different from the original program order with out-of-order execution.

The PowerPC 604 utilizes a branch target address cache and a branch history table to implement branch prediction in the instruction fetch and the decode/dispatch stages, respectively. In the fetch stage, an address is used to access the instruction cache from which up to four instructions can be retrieved. During the instruction cache access, the next fetch address can be predicted by the current fetch address. The current fetch address is used to index into the branch target address cache, which is a fully associative cache with round-robin replacement policy. Each of the 64 branch target address cache entries stores the target address of a predicted taken branch instruction (this can be either a conditional or unconditional branch). A hit in the branch target address cache indicates that in the current fetch group, there is a branch instruction that is predicted as taken. Therefore, that branch’s target address is used as the fetch address in the next cycle. Otherwise, the next sequential fetch address is used.

In the decode/dispatch stage, conditional
branch instructions are re-predicted by the branch history table. The branch history table is a 512-entry, direct-mapped cache that stores the execution history of conditional branches. A two-bit execution history encodes four states: strong not taken, weak not taken, strong taken, and weak taken. Similar to the branch target address cache, the address of a branch instruction is used to access the execution history in the branch history table. If the history is either strong not taken or weak not taken, the branch is predicted not taken. Otherwise, the branch is predicted taken.

An entry in the rename buffer, the reservation stations, and the reorder buffer must be allocated for an instruction before it can be dispatched at the end of the dispatch stage. The rename buffer stores instruction results to avoid contention for a given register file location during out-of-order execution. The PowerPC 604 uses three sets of rename buffers for three register files—a 12-entry general-purpose rename buffer for general-purpose registers, an 8-entry floating-point rename buffer for floating-point registers, and an 8-entry conditional rename buffer for conditional registers.

When an instruction is dispatched to a functional unit, it is placed in a reservation station before execution. Reservation stations allow instructions that do not have ready operands to progress deeper in the pipeline, allowing the front end of the pipeline to process new instructions instead of stalling. The value of each instruction operand is written into the reservation station entry at allocation. If the value is not yet available, the tag (the status bits that identify the result) of the pending operand will be placed in the reservation stations instead of the actual value. Once all operands are available, the instruction is removed from the reservation station and execution begins. The PowerPC 604 has distributed reservation stations, that is, each functional unit has a dedicated two-entry buffer for two instructions of the proper type.

To ensure precise exception handling, out-of-order instructions must complete in program order. By allocating entries in the reorder buffer in the dispatch stage, the program order is preserved in the reorder buffer. When an instruction finishes execution, its status is recorded in the corresponding buffer entry’s status bits. When the instruction finally completes or transfers its speculative state into permanent machine state, the associated reorder buffer entry is deallocated in program order. The PowerPC 604 uses a 16-entry reorder buffer to implement in-order completion. Up to four instructions can retire in a cycle.

Abstract FSM Models

As formulated in prior work, Figure 2 shows the FSM model for each of the 512 branch history table entries. A cold start initializes all entries in the branch history table to the start state, strong not taken. Any conditional branch whose address directly maps to the same branch history table entry will cause transitions in that entry’s FSM when the branch is resolved in the execution stage. The current state of the FSM is used to make predictions (possibly overriding branch target address cache predictions) for instructions in the decode and dispatch stages. States Weak Not Taken and Strong Not Taken make a branch prediction of Not Taken (T = 0) while states Weak Taken and Strong Taken predict taken (T = 1).

Figure 3 shows the FSM model for a branch target address cache entry that corresponds to an unconditional branch. When a new unconditional branch is first encountered, the content of the branch target address cache entry at the position of the round-robin pointer is removed (remove transition) and the branch is predicted not taken (state, T = 0). This is the initialization that allocates branch target address cache entry and loads both the fetch and target address of the branch instruction. After the branch is executed, the FSM transitions to the predicted-taken state. If the same unconditional branch is encountered, it is predicted to be taken (T = 1), and the FSM responds by self-looping on the predicted taken state.

Figure 4 shows a six-state FSM for a branch target address cache entry corresponding to a
A conditional branch. When a new conditional branch enters the branch target address cache, one of the four remove transitions is traversed to a state where a prediction of not taken ($T = 0$) is made. The specific remove transition performed depends on the branch history stored in the corresponding direct-mapped branch history table entry. When the conditional branch is resolved in the execution stage, a transition is made based on the outcome. A branch resolved taken causes a state transition along the resolved taken arc. A resolved not taken outcome causes a resolved not taken transition. The not-taken states (Weak Not Taken and Strong Not Taken) make not-taken predictions ($T = 0$), while the taken states (Weak Taken and Strong Taken) predict branches to be taken ($T = 1$). Future encounters of the branch instruction use the output value indicated in the current state ($T = 0$ or $T = 1$) for the prediction.

The FSM model for the operations of each PowerPC 604 rename buffer entry is shown in Figure 5. A rename buffer entry is free until the dispatch unit allocates an entry for an instruction in the dispatch stage. Allocation occurs if an instruction modifies a register. The entry remains allocated until the instruction completes, and the result is written back to the register file. There are two states for an allocated entry. At the time of renaming, each newly allocated rename entry will always hold the most recent value for the renamed register denoted by the most recent (MR) Allocate state of Figure 5. If a rename entry is allocated to a register, which is then later renamed by another instruction, the previously allocated entry will no longer hold the most recent value and will therefore transition from the MR Allocate state to the non-MR Allocate state.

Once an instruction finishes its execution, its result is written to its corresponding rename entry. The content of the rename entry becomes valid; this causes a transition from MR Allocate (non-MR Allocate) to MR Valid (non-most recent Valid), indicating that the value stored in this entry is valid. The FSM stays in the valid state until the result is written to the register file (WB transition) or a prior instruction causes an exception that requires all subsequent instructions to be discarded (discard transition).
Figure 6 shows the FSM model for each reservation station entry of a two-operand PowerPC instruction (an add instruction for example). Every reservation station entry starts as free and available for allocation. Hence, the starting state for a reservation station entry is the Free state shown in the center of the diagram of Figure 6. When an instruction is in the dispatch stage and all dispatch conditions are met, an entry in the reservation station is allocated. Since the instruction requires two source operands, there are four possible status states for a reservation station entry:

- **Allocate 00**: No source operands are available
- **Allocate 01**: Only the right source is available
- **Allocate 10**: Only the left source is available
- **Allocate 11**: Both source operands are available

The FSM can transitions occur from the Allocate 00, Allocate 01, and Allocate 10 states to another Allocate state when other operand(s) become available. A reservation station entry is deallocated if all operands are ready and the instruction is issued (issue transition); or if the entry is discarded due to an exception created by a prior instruction (discard transition).

A reorder buffer entry is available for allocation if its FSM is in the Free state shown in Figure 7. A transition from the Free state to the Allocate state occurs when the instruction is dispatched to a reservation station. A transition from the Allocate state to the Execute state and finally to the Finish state occurs when the instruction executes and finishes, respectively. The reorder buffer entries can be discarded if the corresponding instructions follow an instruction that causes an exception (discard transition).

### Design Errors

Twenty design errors in the PowerPC 604 presilicon processor model are based on real experiences of industrial designers. Design errors 1 to 10 affect the functionality of the rename buffers, 11 to 14 are reorder buffer errors, 15 to 17 are reservation station errors, and 18 to 20 describe erroneous functionality of the branch prediction mechanism (including both the branch history table and the branch target address cache). In the following, \( N \) is an integer constant that is less than the size of the buffer being considered.

### MTPG Effectiveness

The fMW functional performance simulator is used in our simulation experiments. The existing fMW model of the PowerPC 604 microarchitecture is assumed to be a correct model for the 604’s behavior. Based on this gold model and our buffer-oriented validation methodology, we derive test programs to traverse all of the transitions of the FSM models introduced earlier. Figure 8 gives an overview of the experimental procedure carried out for evaluating MTPG.

At least one instance of each design error type described earlier is instantiated. The design error instances are “injected” into the
1. The logic for mapping the registers to the rename buffer entries fails, such as if instead of reading a general-purpose rename entry for an designed general-purpose register operand, a floating-point rename entry is read.

2. When the destination register is the same as one of the source operands, the rename logic malfunctions. The rename entry of the source operand that is currently being used is allocated for the destination register.

3. When there is an instruction \( i \) using the pending result of another instruction \( j \), the logic for checking this dependency fails. Specifically, the logic incorrectly reports that the rename entry that instruction \( i \) depends on is different from the rename entry allocated for the pending result of instruction \( j \).

4. Validity checking of the rename operation fails. The rename logic incorrectly indicates that the value of an operand is valid although the instruction is still in execution.

5. The rename operation takes \( M \) cycles more than specified.

6. \( N \) entries of the rename buffer are permanently inaccessible. As a result, the rename buffer can exhaust its entries prematurely.

7. When \( N \) entries of the rename buffer are in use, the buffer is considered exhausted. This error is different from the previous one because the inaccessible entries in this case are not permanent. For example, assume the buffer is considered full when \( N=10 \) out of 12 entries are used. In one such case, entry 0 and 1 are inaccessible if entries 2 to 11 are used.

8. The dependency checking fails to identify a stale rename entry. When there are two in-flight instructions, \( i \) preceding \( j \), that modify the same register, the rename buffer logic fails to mark the rename buffer entry allocated for instruction \( i \) to be “non-most recent.” As a result, subsequent instructions following instruction \( i \) and \( j \) will always use the result from the rename buffer with the highest entry number. For example, if the rename entry for instruction \( i \) is entry 11 and the rename entry for instruction \( j \) is entry 0, the result of instruction \( i \) (entry 11) will be used even though it is a stale version.

9. Similar to error 8, two in-flight instructions, \( i \) preceding \( j \), are modifying the same destination register. A subsequent instruction \( k \) uses the result of instruction \( j \). The dependency checking fails when \( k \) is allocating its entry in the rename buffer while \( j \) does not have the result ready (most recent Allocate state) but \( i \) has its result ready (Non-most recent Valid state). The result of instruction \( i \) is used instead of that of instruction \( j \).

10. The PowerPC 604 accesses incorrect conditional rename buffer entries.

11. Let the maximum number of entries that can be retired in a cycle from the reorder buffer be \( M \). For this design error, the reorder buffer only retires instructions less than its capable bandwidth, that is, it deallocates less than \( M \) entries while there are at least \( M \) entries ready to be retired.

12. When the oldest instruction in the reorder buffer finishes execution, the reorder buffer does not retire its instructions immediately (there is an \( M \)-cycle stall before retirement).

13. \( N \) entries of the reorder buffer are permanently inaccessible.

14. When \( N \) entries of the reorder buffer are in use, the buffer is considered full.

15. \( N \) entries in the reservation station are permanently inaccessible.

16. When \( N \) entries of the reservation station are in use, the buffer is considered full.

17. No data can be forwarded from the execution units to the reservation stations. As a result, an instruction waiting for the result of its operands must sit in the reservation station until the result is written back to the designed register.

18. The logic for updating the branch history table states is incorrect. For example, if an entry is in the weak taken state and the branch resolves taken, the entry should transition to the strong taken state. However, it stays in the weak taken state.

19. The round-robin pointer can only access \( N \) entries in the branch target address cache.

20. The branch target address cache fails to remove the branch instruction that should be predicted not taken by the branch history table.
Microarchitecture model of the 604, one at a time. We then use the test programs that are generated specifically for each buffer described previously in the simulation of both the incorrect and error-free models. The percentage of transitions traversed in the error-free model and the number of clock cycles required to execute the given test program are noted at the end of the simulation. Based on this data, the detection of design errors is determined. The error-free model has 100% of its transitions traversed while the incorrect model may or may not have 100% coverage.

Two metrics are used to determine design error detection and coverage. Functional deviation ($\delta_f$) is the discrepancy in the percentage of the transitions traversed between the incorrect and the error-free models. It is calculated by

$$\delta_f = 100\% - TT_{incorrect}$$

where $TT_{incorrect}$ is the percentage of the FSM transitions traversed in the incorrect 604 model. Similarly, timing deviation ($\delta_t$) is the discrepancy in the cycle counts. $\delta_t$ is calculated by

$$\delta_t = CC_{correct} - CC_{incorrect}$$

where $CC_{correct}$ and $CC_{incorrect}$ are the cycle counts of the error-free and incorrect 604 model, respectively. Our test programs effectiveness is based on the coverage of the design errors using these two measures. Functional deviation coverage ($\delta_f$) is the percentage of the injected errors that cause functional deviation, while timing deviation coverage ($\delta_t$) is the percentage of the injected errors that cause timing deviation and errors.

Table 1 shows functional and timing deviations of the design errors. The first column lists all the design errors by the numbers used earlier. The second column shows the number of instances of each simulated design error type. Columns 3 and 4 show the average functional deviation and the number of instances detected by functional deviation for each error type. Similarly, columns 5 and 6 show the average timing deviation and the number of error instances detected. At the bottom of Table 1, functional and timing deviation coverages are reported. Fifty-two out of 63 design error instances are detected by timing deviation, resulting in 83% error coverage ($\delta_t = 83\%$). Functional deviation indicates that 49 errors are detected ($\delta_f = 78\%$). When either metric indicates that there is a discrepancy, the combined error coverage is 98%.

Note that using one metric alone might not be adequate to determine whether the design is error-free. For example, some instances of errors 7, 12, 13 and 14 are not detected by functional deviation. Conversely, timing deviation cannot detect some instances of errors 15, 16, 18 and 19. Both metrics failure to detect error 11 and an instance of error 14 suggests possible limitations of the metrics used or incompleteness of the test programs. All the design errors can be further categorized based on their impact on PowerPC 604 behavior.

Logic-related Errors

Our test programs were effective in detecting this class of errors. The errors included in this class are 1 to 4, 8 to 10, 17, 18, and 20. The simulation results show that all instances of the errors in this class are detected by both functional and timing deviation. The rename buffer and the reservation station errors (1 to 4, 8 to 10, 17, 18, and 17) are detected because they cause an incorrect inter-instruction dependency that replaces the dependency required to traverse specific FSM transitions. The branch prediction
buffer errors (18 to 20) directly cause the FSM models to change, resulting in unreachable FSM transitions.

Timing-related Errors
This class includes errors 5, 11, and 12. All instances of errors 11 and 12 are not detected through functional deviation (as expected) while instances of errors 5 and 12 are all detected through timing deviation. Error 11 is not detected by either metric.

Buffer Access or Allocation Errors
This class includes errors 6, 7, 13 to 16, and 19. Coverage results indicate that the majority of these errors are detected by either functional or timing deviation (or both). However, some instances of errors 7, 13, and 14 are not detected by either metric. These two design errors reveal one weakness of our current approach in that it does not exercise the full capacity of the buffer.

SIMULATION RESULTS show that MTPG-derived test programs can detect design errors that affect both function and performance. All but one of 20 design errors are detected using metrics based on functionality and timing. Analyzing the simulation results indicates many of the design errors considered create incorrect instruction dependencies. Our test programs can detect these errors since a high level of dependency is required to exercise buffer functionality. On the other hand, design errors that affect buffer utilization may not be detected since the test programs only target individual buffer entries. Finally, some design errors only affect performance (cycle count) and not function (such as the final state). Although MTPG is based on functional specifications, it can detect most performance-related errors because of the precise timing needed to create the dependency required to exercise the buffers.

Currently, we are extending BMV to cover greater microarchitecture functionality. For example, an allocation model that describes the access mechanism for a buffer type can be used in conjunction with the per-entry model, such as the FSM model that characterizes each buffer entry like the one in Figure 2. The allocation model indicates the current entry that can be allocated and accessed, and describes how entries can be removed (deallocated). The allocation and per-entry models can also be combined by computing their cross-product.

<table>
<thead>
<tr>
<th>Design error #</th>
<th>No. error instances</th>
<th>Functional deviation</th>
<th>No. instances detected</th>
<th>Timing deviation</th>
<th>No. instances detected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Average df</td>
<td></td>
<td>Average dt</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>12.07</td>
<td>7</td>
<td>7,270</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>97.82</td>
<td>3</td>
<td>49,512</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>63.79</td>
<td>2</td>
<td>31,609</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>32.72</td>
<td>1</td>
<td>13,820</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>1.76</td>
<td>3</td>
<td>16,792</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>22.71</td>
<td>4</td>
<td>730</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>0.4</td>
<td>3</td>
<td>700</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2.07</td>
<td>1</td>
<td>2,277</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>3.33</td>
<td>1</td>
<td>5,090</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>32.20</td>
<td>1</td>
<td>13,444</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>0.00</td>
<td>0</td>
<td>334</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td>0.00</td>
<td>0</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>0.00</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>50.00</td>
<td>4</td>
<td>3,248</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>50.00</td>
<td>4</td>
<td>3,248</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>20.00</td>
<td>1</td>
<td>500</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>7</td>
<td>51.76</td>
<td>7</td>
<td>1,973</td>
<td>6</td>
</tr>
<tr>
<td>19</td>
<td>6</td>
<td>94.73</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>6.67</td>
<td>1</td>
<td>91</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1. Coverage results for the injected design errors.

Functional deviation coverage, \(d_f\) 78%
Timing deviation coverage, \(d_t\) 83%
Combined deviation coverage, \(d_c\) 98%
machine. FSM testing techniques can then be applied to the resulting machine to obtain high-level validation sequences that are easily translated into test programs. We believe the allocation model and cross-product machine can be used not only for specifying microarchitecture features more completely (functionality affected by design error 11 for example) but also support the automation of the BMV methodology.

Acknowledgments
We would like to thank Dr. Pradip Bose at IBM for his discussion on microarchitecture design errors. This research effort is sponsored in part by the Semiconductor Research Corporation under contract DC068.070.

References

Noppaunut U tamaphethai is a PhD candidate in the Department of Electrical and Computer Engineering at Carnegie Mellon University. He received a bachelor’s degree in electrical engineering in 1995 from Brown University and a master’s degree in electrical and computer engineering in 1997 from
Carnegie Mellon University. His research interests include computer architecture, processor verification/validation, and digital systems testing.

Ronald D. (Shawn) Blanton is an associate professor in the Department of Electrical and Computer Engineering at Carnegie Mellon University, where he is a member of the Center for Electronic Design Automation. He received a bachelor’s degree in engineering from Calvin College in 1987, a master’s degree in electrical engineering in 1989 from the University of Arizona, and a PhD degree in computer science and engineering from the University of Michigan, Ann Arbor, in 1995. His research interests include the computer-aided design of VLSI circuits and systems; fault-tolerant computing and diagnosis; verification and testing; and computer architecture. He has worked on the design and test of complex systems with General Motors Research Laboratories, AT&T Bell Laboratories, Intel, and Motorola. Blanton is the recipient of National Science Foundation Career Award and is a member of IEEE and ACM.

John Paul Shen is a professor in the Electrical and Computer Engineering Department of CMU, and director of the Carnegie Mellon Microarchitecture Research Team (CMuART). This fall 2000 he left CMU and joined Intel’s Microprocessor Research Labs (MRL) to head up the Microarchitecture Lab.

For questions regarding this article, please contact Shawn Blanton in the Department of Electrical and Computer Engineering at Carnegie Mellon University, Pittsburgh, PA 15213; e-mail Blanton@ece.cmu.edu.