Specification-based Compaction of Directed Tests for Functional Validation of Pipelined Processors[*]

Heon-Mo Koo[†]
Intel Corporation
1900 Prairie City Road
Folsom, CA 95630, USA
heon-mo.koo@intel.com

Prabhat Mishra
Computer & Information Science & Engineering
University of Florida
Gainesville, FL 32611, USA
prabhat@cise.ufl.edu

ABSTRACT

Functional validation is a major bottleneck in microprocessor design methodology. Simulation is the widely used method for functional validation using billions of random and biased-random test programs. Although directed tests require a smaller test set compared to random tests to achieve the same functional coverage goal, there is a lack of automated techniques for directed test generation. Furthermore, the number of directed tests can still be prohibitively large. This paper presents a methodology for specification-based coverage analysis and test generation. The primary contribution of this paper is a compaction technique that can drastically reduce the required number of directed test programs to achieve a coverage goal. Our experimental results using a MIPS processor and an industrial processor (e500) demonstrate more than 90% reduction in number of directed tests without sacrificing the functional coverage goal.

Categories and Subject Descriptors: B.6.3 [Logic Design]: Design Aids - Verification

General Terms: Design, Verification

Keywords: Test Compaction, Processor Validation

1. INTRODUCTION

Verification complexity is increasing at an alarming rate since it is directly proportional to the design complexity growth of modern processors. A major challenge is how to reduce the overall functional validation effort. In the current industrial practice [1, 16], random and biased-random test program generation at architecture (ISA) level is most widely used for simulation-based validation to uncover errors early in the design cycle. A test program consists of a sequence of instructions. Compared to random or biased-random tests, directed test generation can significantly reduce overall validation effort since the shorter tests can obtain the same coverage goal. The number of directed tests can still be extremely large. Therefore, there is a need for efficient test reduction techniques.

In directed test generation, although a test is generated for activating a particular functional fault, it may go through several pipeline stages and paths over multiple clock cycles to reach the target fault. Therefore, there is a high probability that the test can accompany multiple pipeline interactions before and after it reaches the target functionality. Based on this fact, we present an FSM coverage-directed selection of minimal fault set to reduce the number of functional tests required for validation of pipelined processors. The goal is to generate a reduced set of test vectors without sacrificing the coverage requirement. It is important to note that we perform compaction of required faults before test generation and therefore only need to generate a reduced set of tests. This is in contrast with the existing test compaction approaches, especially in the domain of manufacturing test, where compaction is performed after test generation, therefore, the existing approaches does not reduce the cost of test generation. Our approach can significantly reduce the test generation cost as well as the overall validation effort using the reduced set of functional tests.

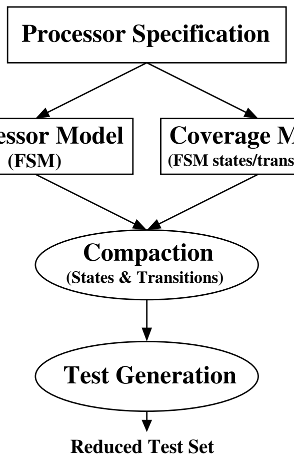

Figure 1: Functional test compaction methodology

Figure 1 shows the overall flow of our fault selection and associated test generation methodology. From specification of a processor, we create a finite state machine (FSM) model and define FSM state and transition coverage metrics based on pipeline interactions. FSM compaction is performed before test generation by eliminating the states and the transitions that are illegal, redundant, or unreachable based on design constraints. The remaining states and transitions are represented in the proposed binary format. One of the un-

[*]This work was partially supported by grants from Intel Corporation and NSF CAREER award 0746261.

[†]This work was performed when the author was a graduate student at the University of Florida.

covered states or transitions is chosen as a target for directed test generation. A path in FSM is traversed from a given point by tracing backward to an initial state and tracing forward to a final state from that point. During the trace process, we select an uncovered transition to minimize test redundancy as well as test volume. The number of uncovered states and transitions are reduced by eliminating the states and transitions on the path. We use model checking to generate a test program to exercise the path. Path selection and test generation continues until all states and transitions are covered. This paper makes three important contributions. First, we propose a simple but efficient FSM model of pipelined processors. Second, we define FSM state and transition coverage based on pipeline interactions. Finally, we propose efficient techniques for fault selection and test generation to reduce overall validation effort.

The rest of the paper is organized as follows. Section 2 presents related work addressing FSM modeling and test compaction in the context of test generation. Section 3 describes our FSM modeling and functional coverage for validation of pipelined processors. Section 4 presents our test compaction technique. Section 5 describes coverage-driven test generation followed by a case study in Section 6. Finally, Section 7 concludes the paper.

2. RELATED WORK

FSM model-based test generation has been developed for validation of pipelined processors where an FSM model is used to generate a test suite based on state, transition, or path coverage [2, 20]. A significant bottleneck in these methods is the high complexity of FSM models, resulting in state explosion problem. To alleviate FSM complexity, abstraction techniques have been proposed [3, 8, 13, 15, 17]. The abstract FSM models provide feasible ways of concrete test generation. Abstract tests are generated from the abstract FSM and then they are converted into test programs. Compared to the existing approaches, our FSM model is easy to create and analyze because its states describe the functional status of each functional unit.

Due to the large volume of test data and the extremely long test time for manufacturing test, considerable research has been done to reduce the structural test data volume. Test compaction techniques are generally categorized into dynamic and static compactions. Dynamic compaction is applied during test generation while static compaction is applied after test generation. Rudnick and Patel [14] have proposed dynamic test compaction for sequential circuits using fault simulation and genetic algorithms. El-Maleh and Osais [5] have presented decomposition-based static compaction algorithms where a test vector is decomposed into atomic components and the test vector is eliminated if its components can be all moved to other test vectors. Set covering has been applied to static compaction procedures for combinational circuits using the fault detection matrix [6, 9]. Dimopoulos and Linardis [4] have modeled static compaction for sequential circuits as a set-covering problem. The matrix reduction techniques [18] can be applied to mitigate the complexity of set covering by eliminating redundant rows (faults) and columns (test vectors) in the fault detection matrix.

A lot of structural test compaction techniques have been proposed in manufacturing test domain because they have a significant impact on overall testing cost and time. There has been no significant progress in functional test compaction in validation domain because functional tests are considered as one-time effort in design methodology. However, millions of tests are used in the current industrial practice, and regression test is conducted every day during design cycle. Therefore, reduction in functional tests will have significant impact on overall design effort by removing redundant tests as well as selecting effective tests.

3. FSM MODEL AND COVERAGE

Our FSM model can be generated from an Architecture Description Language (ADL) [12] specification by defining desired functionalities of a processor using states and their transition functions. The proposed FSM model is used to generate a set of test programs based on FSM coverage metrics. The test set is reusable during design validation at different levels of abstraction including specification and implementation.

3.1 FSM Modeling of Pipelined Processors

An FSM model is defined as $M = (I, O, S, \delta, \lambda)$ where I, O, S, δ, and λ are a finite set of inputs, outputs, states, state transition function $\delta : S \times I \rightarrow S$, and output function $\lambda : S \times I \rightarrow O$, respectively. When the model M is in the state s ($s \in S$) and receives an input a ($a \in I$), it moves to the next state specified by $\delta(s, a)$ and produces an output given by $\lambda(s, a)$. For an initial state s_1, an input sequence $x = a_1, ..., a_k$ takes the M successively to states $s_{i+1} = \delta(s_i, a_i)$, $i = 1, ..., k$ with the final state s_{k+1}. In the pipelined processor FSM model, assuming each a_i corresponds to instruction(s) fetched from instruction cache (or memory), the input instruction sequence $x = a_1, ..., a_k$ can be used as a test program to exercise the path consisting of the states and the state transitions from s_1 to s_{k+1}.

3.1.1 *Modeling of FSM States*

We create the functional states S in the form of binary data from the processor specification that contains both the pipelined structure and the behaviors of the processor. The proposed FSM model is based on interactions among functional units of the pipelined processor. A group of bits are assigned to describe the functional status of each functional unit. A functional state of the entire processor consists of bit concatenation of local states of all functional units. We denote the number of activities in the functional unit fu_j by r_j and the number of bits to be assigned to the unit by b_j where $j = 1, ..., U$ and U is the number of functional units in the processor. Therefore, the total number of bits to describe the processor FSM states is $N = \sum_{j=1}^{U} b_j$. We denote the number of states in the machine M by $NS = |S| = 2^N$. The state of functional unit fu_j is denoted as ss_j using b_j bits and the state s_k of the processor FSM can be defined by concatenating ss_1, ss_2, ..., ss_U.

For example, we assign two bits to represent four functional states of Fetch (IF) unit: '00' for idle, '01' for normal operation (instruction fetch), '10' for stall, and '11' for exception. Figure 2 shows an example of the FSM states of the pipelined processor. Given that all the functional units have only four possible states, each unit requires 2 bits for its four functionalities. This binary format of functional FSM model provides an efficient indexing mechanism to access and analyze each functional state. In addition, next states can be described as Boolean functions. For example, assuming the

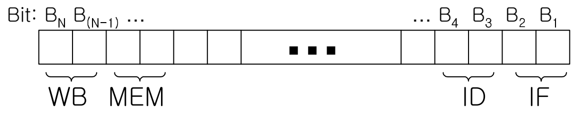

Figure 2: Binary format of the states in FSM model

state transitions (s_i, s_j) and (s_i, s_k) with s_j = '0011' and s_k = '0010', the next states of s_i are expressed as $\bar{B}_4\bar{B}_3B_2B_1 + \bar{B}_4\bar{B}_3B_2\bar{B}_1 = \bar{B}_4\bar{B}_3B_2$. For each state, a corresponding index number has a list of the next and previous states that are produced using transition functions. The list of neighboring states are used for selecting a uncovered path during test generation.

3.1.2 Modeling of FSM Transitions

The state transition functions are based on pipeline behaviors of processors. The pipeline behaviors are the rules in each pipeline stage that determine when instructions can move to the next stage and when they cannot. For pipeline behavior modeling, we decompose the entire processor FSM into smaller FSMs at functional unit level. Since not all the functional units affect the next states of other functional units, the transition functions of the FSM can be decomposed into sub-functions each of which is dedicated to a specific functional unit.

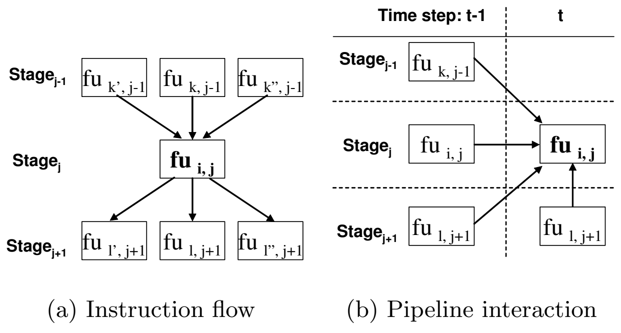

Figure 3: Pipeline behavior

Figure 3 shows the general behaviors of pipelined processors. Each instruction goes through the current pipeline stage to the next stage as shown in Figure 3(a), where fu is a functional unit, $1 \leq i, k, l \leq U$, $1 \leq j \leq D$, and D is the pipeline depth. Each functional unit $fu_{i,j}$ can interact with different number of functional units at stage $j-1$ and $j+1$. For example, a decode unit may have multiple execution units at its following stage while a fetch unit typically has only one unit (decode unit) at the following stage.

Figure 3(b) shows the pipeline interaction of the functional unit $fu_{i,j}$. The state of $fu_{i,j}$ at time step t is decided by the previous and current states of units $fu_{k,j-1}$ and $fu_{l,j+1}$ as well as itself. For example, if $fu_{l,j+1}$ and $fu_{i,j}$ are on the same pipeline and $fu_{l,j+1}$ is in the stall state at time step t, then $fu_{i,j}$ should be in stall state because the instruction in $fu_{i,j}$ cannot go to the next stage $fu_{l,j+1}$. Considering feedback loop such as data forwarding in the pipelined processor, $fu_{i,j}$ at time t will be affected by the state of $fu_{i,j+\alpha}$ at $(t-1)$ where $0 \leq \alpha \leq D$.

Based on the pipeline behavior, the state transition to the functional unit $fu_{i,j}$ at time step t is defined as $ss_{i,j}(t) = f(ss_{k,j-1}(t-1), ss_{i,j}(t-1), ss_{l,j+1}(t-1), ss_{l,j+1}(t))$. Here, $ss_{i,j}(t)$ represents a set of bits to describe the functional state of $fu_{i,j}$ at time t, and f represents a transition function decided by unit interactions. Therefore, the state s of the processor FSM can be expressed by concatenating $ss_{i,j}$ where i = 1, ..., U and $1 \leq j \leq D$.

3.2 Functional Coverage

State coverage and transition coverage are used as coverage metrics to generate a test set. State coverage ensures that every state of an FSM has been visited. Transition coverage ensures that every transition between FSM states has been traversed.

The state coverage of the proposed FSM model is identical to the pipeline interaction coverage that tries to detect whether a set of pipeline interactions (between functional units) have been activated at a given clock cycle. Because each FSM state consists of multiple sub-states of each functional unit in the processor. Therefore, a test program that covers the FSM state will activate the corresponding pipeline interaction. We can compute the number of theoretically possible FSM states based on the number of functional units and the number of activities at each unit. In general, the number of activities varies for different units depending on what activities we want to test, thereby each unit may require different number of bits for its functional states. Considering an FSM model with m units where each unit has on average r activities, the FSM will have r^m states which can be extremely large even for small number of activities. For example, a MIPS processor [7] with 17 functional units and 4 activities has approximately seventeen billion states.

From the point of functionality, the proposed state transition coverage represents temporal pipeline interactions. Based on the state transition functions, each state has a list of their next states. When a test visits the state and goes to one of its next state, we put the next state off the list since the transition between the two states is covered. State transition coverage of the FSM is achieved when the next state lists are empty for every states. The number of state transitions is determined by the processor's functional behaviors. Theoretically, the maximum number of state transitions is N^2, where N is the number of states, assuming any state can be reached from another state in one step. This theoretical large number of functional states and transitions can be reduced by eliminating unreachable states using functional constraints described in the processor specification.

4. COMPACTION OF FSM MODEL

The state and transition compaction of an FSM plays a major role in efficient test generation since reduction of one state or transition implies one less test vector to generate and apply on RTL implementation. The basic idea is to identify and eliminate all the unreachable and redundant states as well as transitions with respect to coverage-driven test generation.

4.1 Identifying Unreachable States

We use functional constraints described in the processor specification to distinguish unreachable states from reachable ones. The constraints are represented as binary patterns of FSM states. The states with these patterns are removed from the FSM and they are not considered during test generation and coverage analysis since they are unreachable. For example, assume that decode (ID) unit has the single instruction issue constraint and there are two parallel execution units *EX1* and *EX2* in the next pipeline stage.

Since only one instruction can be passed to either *EX1* or *EX2*, both execution units cannot be in normal operation (executes a valid instruction) at the same time. Assuming that *EX1* and *EX2* correspond to the state variables B_6B_5 and B_4B_3 respectively in 8-bit FSM processor state model, the binary pattern 'xx0101xx' represents unreachable states for the single issue constraint, where '01' represents the unit state of normal operation and 'x' represents '0' or '1'. By applying all the functional constraints described in the processor specification, we can identify the unreachable states in the FSM and compute the number of reachable states.

4.2 Identifying Illegal State Transitions

Extracting FSM state transitions at the processor level is very difficult since specification documents do not include relation of processor-level states in general. However, the processor specification provides the rules in each pipeline unit about when instructions can move to the next stage and when they cannot. These pipeline behaviors are used to identify illegal state transitions. We also leverage the decomposition of a processor state transition into functional unit level transitions. For example, if the state of a functional unit $ss_{i,j}$ is in normal operation at time t, then the state of the unit in the previous stage ($ss_{k,j-1}$) cannot be in idle state at time $t-1$ since the instruction in $fu_{i,j}$ must be ready at the previous pipeline stage at time $t-1$.

Table 1: Transition rules between $ss_{k,j-1}$ and $ss_{i,j}$

$ss_{k,j-1}(t-1)$	$ss_{i,j}(t)$
idle	idle, stall
normal op.	normal op., stall, exception
stall	idle, stall
exception	idle, stall

Table 2: Transition rules between $ss_{i,j}$ and $ss_{i,j}$

$ss_{i,j}(t-1)$	$ss_{i,j}(t)$
idle	idle, normal op., stall, exception
normal op.	idle, normal op., exception
stall	idle, normal op., stall, exception
exception	idle

Table 3: Transition rules between $ss_{l,j+1}$ and $ss_{i,j}$

$ss_{l,j+1}(t-1)$	$ss_{i,j}(t)$
idle	idle, normal op., stall, exception
normal op.	idle, normal op., stall, exception
stall	idle, normal op., stall, exception
exception	idle

Sub-state transition rules between units are shown above assuming four functional activities at each unit and one register between consecutive pipeline stages. For example, in Table 1, if $ss_{k,j-1}(t-1)$ = stall, then $ss_{i,j}(t)$ can be either in idle or stall state because no instruction moves from the previous stage. In Table 2 and Table 3, if $ss_{k,j-1}(t-1)$ or $ss_{l,j+1}(t-1)$ = exception, then $ss_{i,j}(t)$ should be in idle state to flush the following instructions in the pipeline.

4.3 Identifying Redundancy

We define redundant states and transitions in terms of coverage-driven test generation. A state (transition) is redundant if the test generated for activating any other states or transitions has to go through this state (transition). This redundant state (transition) is called an *inevitable state* (*transition*). Identifying a redundant state (transition) is similar to finding fault dominance in manufacturing test compaction except the fact that in this case we do not need the generated tests.

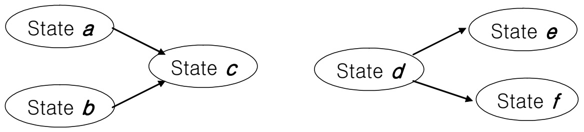

Figure 4: Single transitions between states

We employ various techniques to remove redundant states and transitions. Figure 4 shows inevitable states and transitions that have single outgoing transition (from states a and b) and single incoming transition (to states e and f). The state c is an inevitable state because all the paths from a and b should include the state c. Similarly, the state d is an inevitable state because all the paths to e and f should include the state d. The transitions ($a \rightarrow c$), ($b \rightarrow c$), ($d \rightarrow e$), and ($d \rightarrow f$) are inevitable transitions to their neighbors. We can eliminate the test cases that activate these inevitable states and transitions since any test program that exercises their neighboring states will also activate the inevitable states. The next state lists of each state are used to identify the inevitable states of the single outgoing transitions. If a state has only one state in its next state list, the next state is an inevitable state. In the same way, the previous state lists are used to identify the single incoming transitions.

5. TEST GENERATION

FSM coverage metrics provide a mechanism to evaluate verification progress. In FSM coverage-driven test generation, tests are created to activate a target coverage point (state or transition).

5.1 Coverage-driven Test Selection

Once all the unreachable and redundant tests are removed, one of the uncovered states or transitions is chosen as a target for directed test generation. Each state (binary index number) has a flag to indicate whether the state is covered or not. The flag is called *StateCovered* flag and is initialized to 0. Each state also has a list of its neighboring states, i.e., a list of its transitions. In the list, each neighboring state has two flags to indicate whether the state is a next state or a previous state and whether the transition to/from the neighboring state is covered or not. The second flag is called *TransitionCovered* flag and is initialized to 0.

Beginning from a target state, we search for an FSM path that can cover maximum number of states and transitions. For backward path, one of the previous neighboring states is selected that has *TransitionCovered*=0. A pair of state and transition is covered by setting value 1 for *StateCovered* of the current state and *TransitionCovered* of the previous state. This process continues until the path traversal reaches an initial state. If all of the previous states in the neighbor list are covered (*TransitionCovered*=1) at the current state, we check the neighbor list of the previous states to determine whether the path includes any uncovered state and transition. We extend the search space until an uncovered state/transition is encountered, or until the number of backward transitions reaches its upper bound (maximum number of clock cycles in which an instruction can stay in the processor pipeline). Similarly, we complete the path by tracing forward to a final state from the target point. We continue path generation until all states and transitions in the FSM are covered, i.e., all of the *StateCovered* and *TransitionCovered* flags are set to 1.

5.2 Directed Test Generation

We use model checking for directed test generation because of its capability of automatically producing a counterexample. Figure 5 shows a test generation framework using model checking [10, 11]. In this scenario, a desired behavior is expressed in the form of temporal logic property. A model checker exhaustively searches all reachable states of the processor model to check if the property holds (*verification*) or not (*falsification*). If the model checker finds any reachable state that does not satisfy the property, it produces a counterexample. This falsification can be quite effectively exploited for test generation. Instead of a desired property, its negated version is applied to model checking. A model checker produces a counterexample and corresponding input requirements. The input requirements of the processor model contains a sequence of instructions from an initial state to the point where the negated version of the property fails.

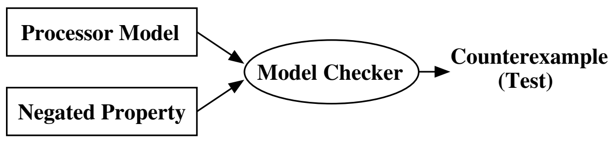

Figure 5: Test generation using model checking

For example, a path selected in the previous section is described in the form of a temporal logic property EFp (i.e., *there exists a path p in the processor model*) where p consists of the states at every time step on the path. Its negated property $AG\neg p$ (i.e., *there is no such path p*) is applied to the model checker. The model checker generates a counterexample path with an input instruction sequence. Since this counterexample path is the same as the selected path, the instruction sequence can be used as a test program to exercise the selected path.

6. EXPERIMENTS

We applied our test compaction methodology on a single-issue MIPS architecture [7] and a superscalar (dual-issue) e500 processor [19]. Figure 6 shows the MIPS architecture. There are 17 functional units. We consider four functional states (activities) of each unit: '00' for idle, '01' for normal operation, '10' for stall, and '11' for exception. Figure 7 shows the functional FSM model of the processor in the form of 33-bit binary. We assume that WB has only two states (idle and normal operation), and IALU and DIV have the exception state for overflow and divide-by-zero, respectively. All other functional units have three states (idle, normal operation, and stall). In the figure, the term "oper" represents "normal operation". Therefore, theoretically possible number of states is $2 \times 4 \times 4 \times 3^{14} \simeq 153 \times 10^6$.

Unreachable states are removed by using the constraints of processor behavior such as single issue requirement. For example, the unreachable binary pattern 'xxxx...0101xxxx'

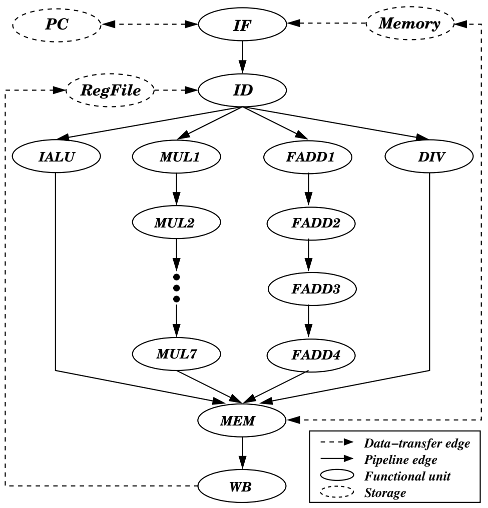

Figure 6: A MIPS architecture

(where x is a don't-care bit) represents the single issue constraint that two execution units IALU and MUL1 cannot be in normal operation at the same clock cycle. The corresponding number of states is 12.7×10^6. We can eliminate those states in the FSM model since the states with this pattern are not allowed due to single-issue constraint. After removing all unreachable states, the number of states is reduced to 87.2×10^6 (43% reduction).

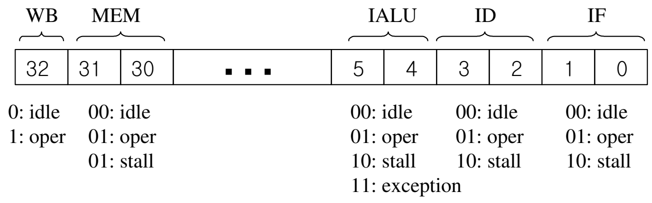

Figure 7: 33-bit FSM state model

Table 4 presents the results of our test compaction for the MIPS processor. The first column indicates various compaction steps. The second and third column present compaction results for states and transitions respectively. For example, the value 87.2×10^6 (in second row, second column) indicates the total number of reachable states after performing reachability analysis (on original 153×10^6 states). For FSM state compaction, we searched for incoming and outgoing single transitions that have inevitable neighboring states. We do not need to generate a test for those states since the test programs to exercise their neighbors will cover them.

Table 4: Test compaction results for MIPS processor

	Compaction of FSM States	Compaction of Transitions
Reachable States/Transitions	87.2×10^6	693.1×10^6
Legal and Required States/Transitions	83.5×10^6	376.3×10^6
Selected Tests (States/Transitions)	16.9×10^6	
Overall Reduction	**97.8%**	

After state compaction, we could reduce the number of tests by 3.7×10^6 (4% reduction). As a result, the number of directed test programs before test selection is 83.5×10^6. The FSM transition coverage needs to activate 376.3×10^6 transitions. Our framework of selecting tests (minimum number of states/transitions required to achieve 100% state and transition coverage) produced 16.9×10^6 test programs. Therefore, our approach generates 97.8%[1] overall reduction of directed tests without sacrificing the functional coverage goal.

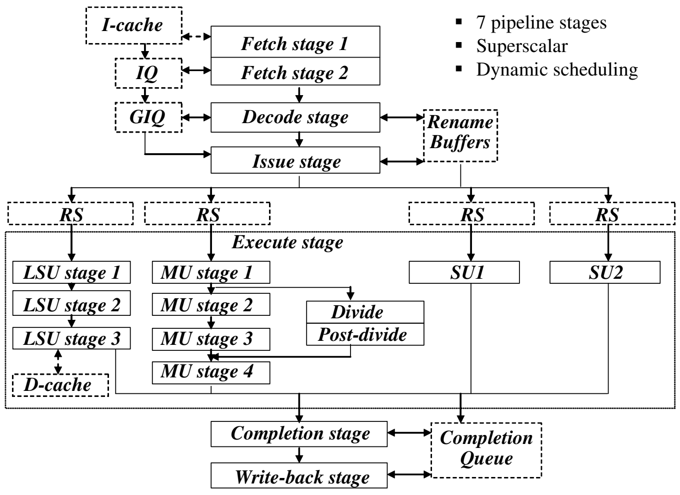

Figure 8: Instruction pipeline flow of e500 processor

We also applied our test generation methodology on a simplified version of the dual-issue e500 processor based on the Power Architecture^{TM} Technology[2] [19]. Figure 8 shows the dual-issue e500 architecture. It has seven pipeline stages and 15 functional units. We used the 29-bit FSM model for the e500 processor with the same assumption as the MIPS processor. Theoretically possible number of states is 22.7×10^6. In our experiment, the e500 processor has less number of states compared to the MIPS processor because we modeled 15 functional units by assuming only single instruction buffer between neighboring pipeline stages. Table 5 presents the results of test compaction for e500 architecture. Our approach generates 92.6% overall reduction in directed tests.

Table 5: Test compaction results for e500 processor

	Compaction of FSM States	Compaction of Transitions
Reachable States/Transitions	14.6×10^6	135.7×10^6
Legal and Required States/Transitions	14.2×10^6	134.6×10^6
Selected Tests (States/Transitions)	11.1×10^6	
Overall Reduction	**92.6%**	

7. CONCLUSIONS

Functional verification is a major bottleneck in pipelined processor design methodology. Simulation is widely used for processor validation using billions of random and constrained-random tests. The directed tests can significantly reduce overall validation effort since directed tests require a smaller test set compared to random tests to achieve the same functional coverage goal. However, the number of directed tests can be still extremely large – in the order of millions.

This paper presented a functional test compaction technique that can significantly reduce the number of directed tests without sacrificing the functional coverage. This paper made three important contributions. First, it developed an efficient FSM model of pipelined processors and defined FSM state and transition coverage based on pipeline interactions. Second, we developed an FSM compaction technique to eliminate redundant states/transitions that can be covered by the remaining states/transitions with respect to test generation. Finally, we developed efficient techniques for fault selection and test generation to reduce overall validation effort. Our experimental results using MIPS and e500 processors demonstrated more than 90% reduction in functional tests without sacrificing the coverage goal.

[1]$(87.2+376.3-16.9)/(87.2+376.3)=97.8\%$

[2]The Power Architecture and Power.org wordmarks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org

8. REFERENCES

[1] A. Adir et al. Genesys-pro: Innovations in test program generation for functional processor verification. *Design & Test*, 21(2):84–93, 2004.

[2] D. Campenhout et al. High-level test generation for design verification of pipelined microprocessors. *DAC*, 1999.

[3] K.-T. Cheng et al. Automatic generation of functional vectors using the extended finite state machine model. *ACM TODAES*, 1(1):57–79, 1996.

[4] M. Dimopoulos and P. Linardis. Efficient static compaction of test sequence sets through the application of set covering techniques. *DATE*, 194–199, 2004.

[5] A. El-Maleh and Y. Osais. Test vector decomposition-based static compaction algorithms for combinational circuits. *ACM TODAES*, 8(4):430–459, 2003.

[6] P. Flores et al. On applying set covering models to test set compaction. *GLSVLSI*, 8–11, 1999.

[7] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2002.

[8] R. Ho et al. Architecture validation for processors. *ISCA*, 404–413, 1995.

[9] D. Hochbaum. An optimal test compression procedure for combinational circuits. *TCAD*, 15(10):1294–1299, 1996.

[10] H.-M. Koo and P. Mishra. Functional test generation using property decompositions for validation of pipelined processors. *DATE*, 1240–1245, 2006.

[11] P. Mishra and N. Dutt. Specification-driven directed test generation for validation of pipelined processors. *ACM TODAES*, 13(2), article 42, 36 pages, 2008.

[12] P. Mishra and N. Dutt, Editors. *Processor Description Languages.* Morgan Kaufmann, 2008.

[13] D. Moundanos et al. Abstraction techniques for validation coverage analysis and test generation. *Computers*, 47(1):2–14, 1998.

[14] E. Rudnick et al. Efficient techniques for dynamic test sequence compaction. *Computers*, 48(3), 1999.

[15] J. Shen and J. A. Abraham. An RTL abstraction technique for processor microarchitecture validation and test generation. *JETTA*, 16(1-2):67–81, 2000.

[16] K. Shimizu et al. Verification of the cell broadband engine processor. *DAC*, 338–343, 2006.

[17] N. Utamaphethai et al. Effectiveness of microarchitecture test program generation. *Design & Test*, 17(4):38–49, 2000.

[18] T. Villa et al. Explicit and implicit algorithms for binate covering problems. *IEEE TCAD*, 16(7):677–691, 1997.

[19] Freescale PowerPc e500 core family reference manual. *http://www.phxmicro.com/Online/E500CORERM.pdf.*

[20] Y. Zhang et al. Using model-based test program generator for simulation validation. *ESS*, (3605):549–556, 2005.