Functional Coverage Driven Test Generation for Validation of Pipelined Processors*

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

Nikil Dutt
Center for Embedded Computer Systems
Donald Bren School of Information and Computer Sciences
University of California, Irvine, CA 92697, USA
dutt@uci.edu

Abstract

Functional verification of microprocessors is one of the most complex and expensive tasks in the current system-on-chip design process. A significant bottleneck in the validation of such systems is the lack of a suitable functional coverage metric. This paper presents a functional coverage based test generation technique for pipelined architectures. The proposed methodology makes three important contributions. First, a general graph-theoretic model is developed that can capture the structure and behavior (instruction-set) of a wide variety of pipelined processors. Second, we propose a functional fault model that is used to define the functional coverage for pipelined architectures. Finally, test generation procedures are presented that accept the graph model of the architecture as input and generate test programs to detect all the faults in the functional fault model. Our experimental results on two pipelined processor models demonstrate that the number of test programs generated by our approach to obtain a fault coverage is an order of magnitude less than those generated by traditional random or constrained-random test generation techniques.

1 Introduction

As embedded systems continue to face increasingly higher performance requirements, deeply pipelined processor architectures are being employed to meet desired system performance. Functional validation of such programmable processors is one of the most complex and expensive tasks in the current Systems-on-Chip (SOC) design methodology. Simulation is the most widely used form of microprocessor verification: millions of cycles are spent during simulation using a combination of random and directed test cases in traditional validation flow. Several coverage measures are commonly used, such as code coverage, toggle coverage and fault coverage. Unfortunately, these measures do not have any direct relationship to the functionality of the device. For example, none of these determine if all possible interactions of hazards, stalls and exceptions are tested in a processor pipeline. Thus there is a need for a coverage metric based on the functionality of the design.

To define a useful functional coverage metric, we need to define a fault model of the design that is described at the functional level and independent of the implementation details. In this paper, we present a functional fault model for pipelined processors. The fault model should be applicable to the wide varieties of today's microprocessors from various architectural domains (such as RISC, DSP, VLIW and Superscalar) that differ widely in terms of their structure (organization) and behavior (instruction-set). We have developed a graph-theoretic model that can capture a wide spectrum of pipelined processors, coprocessors, and memory subsystems. We have defined functional coverage based on the effects of faults in the fault model applied at the level of the graph-theoretic model. This allows us to compute functional coverage of a pipelined processor for a given set of random or constrained-random test sequences.

We have developed test generation procedures that accept the graph model of the pipelined processor as input and generate test programs to detect all the faults in the functional fault model. We applied our methodology on two pipelined processors: a VLIW implementation of the DLX architecture [5], and a RISC implementation of the SPARC V8 architecture [19]. Our experimental results demonstrate two important aspects of our technique. First, it shows how our functional coverage can be used in an existing validation flow that uses random or directed-random test programs. Second, it demonstrates that the required number of test sequences generated by our algorithms to obtain a given fault (functional) coverage is an order of magnitude less than the random or constrained-random test programs.

The rest of the paper is organized as follows. Section 2 presents related work addressing validation of pipelined processors. Section 3 presents a graph-based modeling of pipelined architectures. The functional fault models are described in Section 4. Section 5 defines the functional coverage based on the fault model. Section 6 presents the test generation procedures followed by a case study in Section 7. Finally, Section 8 concludes the paper.

2 Related Work

Traditionally, validation of a microprocessor has been performed by applying a combination of random and directed test programs using simulation techniques. Many techniques have been proposed for generation of directed test programs [1, 4, 9, 11, 12]. These techniques do not consider pipeline behavior for generating test programs.

Ur and Yadin [16] have presented a method for generation of assembler test programs that systematically probe the micro-architecture of a PowerPC processor. Iwashita et al. [7] use an FSM based processor modeling to automatically generate test programs. Campenhout et al. [3] have proposed a test genera-

*This work was partially supported by NSF grants: CCR-0203813 and CCR-0205712.

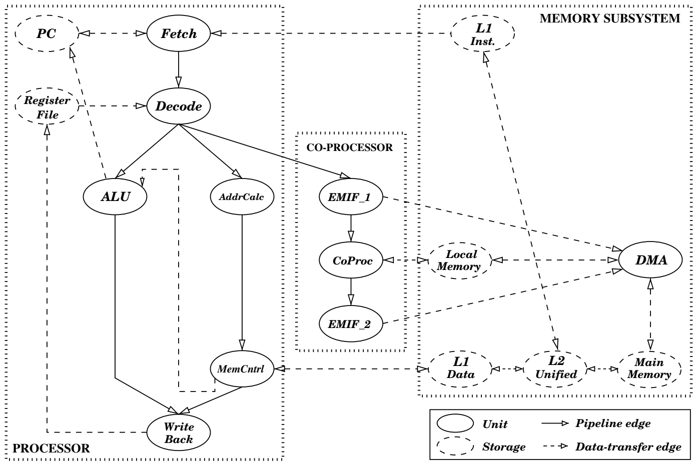

Figure 1. A Structure Graph of a Simple Architecture

tion algorithm that integrates high-level treatment of the datapath with low-level treatment of the controller. Kohno et al. [8] have presented a tool that generates test programs for verifying pipeline behavior in the presence of hazards and exceptions. Ho et al. [6] have presented a technique for generating test vectors for verifying the corner cases of the design. Mishra et al. [10] have proposed a graph-based functional test program generation technique for pipelined processors using model checking. None of these techniques provides a comprehensive metric to measure the coverage of the pipeline interactions. An extensive survey on coverage metrics in simulation-based verification is presented by Tasiran et al. [13]. Andrew Piziali [2] has presented a comprehensive study on functional verification coverage measurement and analysis.

Many researchers have proposed techniques for generation of functional test programs for manufacturing testing of microprocessors ([14], [15]). These techniques use stuck-at fault coverage to demonstrate the quality of the generated tests. To the best of our knowledge, there are no previous approaches that describe functional fault models for pipelined architectures, use it to define functional coverage, and generate test programs to detect all the functional faults in the fault model.

3 Architecture Model of a Pipelined Processor

Modeling plays a central role in the generation of test programs for validation of pipelined processors. In this section, we briefly describe how the graph model captures the structure and behavior of the processor using the information available in the architecture manual.

3.1 Structure

The structure of an architecture pipeline is modeled as a graph with the components as nodes and the connectivity as the edges. We consider two types of components: *units* (e.g., ALUs) and *storages* (e.g., register files). There are two types of edges: *pipeline edges* and *data transfer edges*. A pipeline edge transfers instruction (operation) between two units. A data-transfer edge transfers data between units and storages.

For illustration, we use a simple multi-issue architecture consisting of a processor, a co-processor and a memory subsystem. Figure 1 shows the graph-based model of this architecture that can issue up to three operations (an ALU operation, a memory access operation, and a coprocessor operation) per cycle. In the figure, oval boxes denote units, dotted ovals are storages, bold edges are pipeline edges, and dotted edges are data-transfer edges. A path from a root node (e.g., Fetch) to a leaf node (e.g, WriteBack) consisting of units and pipeline edges is called a *pipeline path*. For example, one of the pipeline path is $\{$*Fetch, Decode, ALU, WriteBack*$\}$. A path from a unit to main memory or register file consisting of storages and data-transfer edges is called a *data-transfer path*. For example, $\{$*MemCntrl, L1, L2, MainMemory*$\}$ is a data-transfer path.

3.2 Behavior

The behavior of the architecture is typically captured by the instruction-set (ISA) description in the processor manual. It consists of a set of operations[1] that can be executed on the architecture. Each operation in turn consists of a set of fields (e.g. opcode, arguments) that specify, at an abstract level, the execution semantics of the operation. We model the behavior as a graph, where the nodes represent the fields of each operation and the edges represent orderings between the fields. Figure 2 describes a portion of the behavior (consisting of two operation graphs) for the example processor shown in Figure 1.

Nodes are of two types: opcode and argument. The opcode nodes represent the opcode (i.e. mnemonic), and the argument nodes represent argument fields (i.e., source and destination arguments). In Figure 2, the ADD and STORE nodes are opcode nodes, while the others are argument nodes. Edges are also of two types: operation and execution. The operation edges link the fields of the operation and also specify the syntactical ordering between them. On the other hand, the execution edges specify the execution ordering between the fields. In Figure 2, the solid edges represent operation edges while the dotted edges represent execution edges. For the ADD operation, the operation edges specify that the syntactical ordering is opcode followed by DEST, SRC1 and SRC2 arguments (in that order), and the execution edges specify that the SRC1 and SRC2 arguments are executed (i.e., read) before the ADD operation is performed. Finally, the DEST argument is written.

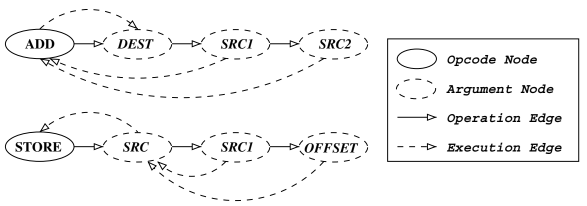

Figure 2. A Fragment of the Behavior Graph

The architecture manual also provides information regarding the mapping between the structure and behavior. We define a set of mapping functions that map nodes in the structure to nodes in the behavior (and vice-versa). The *unit-to-opcode (opcode-to-unit)* mapping is a bi-directional function that maps unit nodes in the structure to opcode nodes in the behavior. The *unit-to-opcode* mappings for the architecture in Figure 1

[1]In this paper we use the terms operation and instruction interchangeably.

include mappings from *Fetch* unit to opcodes {*ADD, STORE*}, *ALU* unit to opcode *ADD*, *AddrCalc* unit to opcode *STORE* etc. The *argument-to-storage (storage-to-argument)* mapping is a bi-directional function that maps argument nodes in the behavior to storage nodes in the structure. For example, the *argument-to-storage* mappings for the *ADD* operation are mappings from {*DEST, SRC1, SRC2*} to *RegisterFile*.

4 Functional Fault Models

In this section, we present fault models for various functions in a pipelined processor. We categorize various computations in a pipelined processor into *register read/write*, *operation execution*, *execution path* and *pipeline execution*. We outline the underlying fault mechanisms for each fault model, and describe the effects of these faults at the level of the architecture model presented in Section 3.

4.1 Fault Model for Register Read/Write

To ensure fault-free execution, all registers should be written and read correctly. In the presence of a fault, reading of a register will not return the previously written value. The fault could be due to an error in reading, register decoding, register storage, or prior writing. The outcome is an unexpected value. If V_{R_i} is written in register R_i and read back, the output should be V_{R_i} in fault-free case. In the presence of a fault, output $\neq V_{R_i}$.

4.2 Fault Model for Operation Execution

All operations must execute correctly if there are no faults. In the presence of a fault, the output of the computation is different from the expected output. The fault could be due to an error in operation decoding, control generation or final computation. Erroneous operation decoding might return an incorrect opcode. This can happen if incorrect bits are decoded for the opcode. Selection of incorrect bits will also lead to erroneous decoding of source and destination operands. Even if the decoding is correct, due to an error in control generation an incorrect computation unit can be enabled. Finally, the computation unit can be faulty. The outcome is an unexpected result. Let val_i, where $val_i = f_{opcode_i}(src_1, src_2, ...)$, denote the result of computing the operation "$opcode_i\ dest,\ src_1,\ src_2,\ ...$". In the fault-free case, the destination will contain the value val_i. Under a fault, the destination is not equal to val_i.

4.3 Fault Model for Execution Path

During execution of an operation in the pipeline, one pipeline path and one or more data-transfer paths get activated. We define all these activated paths as the *execution path* for that operation. An execution path ep_{op_i} is faulty if it produces incorrect result during execution of operation op_i in the pipeline. The fault could be due to an error in one of the paths (pipeline or data-transfer) in the execution path. A path is faulty if any one of its nodes or edges are faulty. A node is faulty if it accepts valid inputs and produces incorrect outputs. An edge is faulty if it does not transfer the data/instruction correctly.

Without loss of generality, let us assume that the processor has p pipeline paths ($PP = \cup_{i=1}^{p} pp_i$) and q data-transfer paths ($DP = \cup_{j=1}^{q} dp_j$). Furthermore, each pipeline path pp_i is connected to a set of data-transfer paths $DPgrp_i$ ($DPgrp_i \subseteq DP$). During execution of an operation op_i in the pipeline path pp_i, a set of data-transfer paths DP_{op_i} ($DP_{op_i} \subseteq DPgrp_i$) are used (activated). Therefore, the execution path ep_{op_i} for operation op_i is, $ep_{op_i} = pp_i \cup DP_{op_i}$. Let us assume, operation op_i has one opcode ($opcode_i$), m sources ($\cup_{j=1}^{m} src_j$) and n destinations ($\cup_{k=1}^{n} dest_k$). Each data-transfer path dp_i ($dp_i \in DP_{op_i}$) is activated to read one of the sources or write one of the destinations of op_i in execution path ep_{op_i}. Let val_i, where $val_i = f_{opcode_i}(\cup_{j=1}^{m} src_j)$, denote the result of computing the operation op_i in execution path ep_i. The val_i has n components ($\cup_{k=1}^{n} val_i^k$). In the fault-free case, the destinations will contain correct values, i.e., $\forall k\ dest_k = val_i^k$. Under a fault, at least one of the destinations will have incorrect value, i.e., $\exists k\ dest_k \neq val_i^k$.

4.4 Fault Model for Pipeline Execution

The previous fault models consider only one operation at a time. An implementation of a pipeline is faulty if it produces incorrect results due to execution of multiple operations in the pipeline. The fault could be due to incorrect implementation of the pipeline controller. The faulty controller might have erroneous hazard detection, incorrect stalling, erroneous flushing, or wrong exception handling schemes.

Let us define stall set for a unit u (SS_u) as all possible ways to stall that unit. Therefore, the stall set for the architecture $StallSet = \cup_{\forall u} SS_u$. Let us also define the exception set for a unit u (ES_u) as all possible ways to create an exception in that unit. We define the set of all possible multiple exception scenarios as $MESS$. Hence, the exception set for the architecture $ExceptionSet = \cup_{\forall u} ES_u \cup MESS$. We consider two types of pipeline interactions: stalls and exceptions. Therefore, all possible pipeline interactions (PIs) can be defined as: $PIs = StallSet \cup ExceptionSet$. Let us assume a sequence of operations ops_{pi} causes a pipeline interaction pi (i.e., $pi \in PIs$), and updates n storage locations. Let val_{pi} denote the result of computing the operation sequence ops_{pi}. The val_{pi} has n components ($\cup_{k=1}^{n} val_{pi}^k$). In the fault-free case, the destinations will contain correct values, i.e., $\forall k\ dest_k = val_i^k$. Under a fault, at least one of the destinations will have incorrect value, i.e., $\exists k\ dest_k \neq val_i^k$.

5 Functional Coverage Estimation

We define functional coverage based on the fault models described in Section 4. Consider the following cases:

- a fault in *register read/write* is covered if the register is written first and read later.
- a fault in *operation execution* is covered if the operation is performed, and the result of the computation is read.
- a fault in *execution path* is covered if the execution path is activated, and the result of the computation is read.
- a fault in *pipeline execution* is covered if the fault is activated due to execution of multiple operations in the pipeline, and the result of the computation is read.

We compute functional coverage of a pipelined processor for a given set of test programs as the ratio between the number of faults detected by the test programs and the total number of detectable faults in the fault model.

6 Test Generation Techniques

In this section, we present test generation procedures for detecting faults covered by the fault models presented in Section 4. Different architectures have specific instructions to observe the contents of registers and memories. In this paper, we use load and store instructions to make the register and memory contents observable at the output data bus.

We first describe a procedure *createTestProgram* that is used by the test generation algorithms. The procedure accepts a list of operations as input and returns a modified list. It assigns appropriate values to the unspecified locations (opcodes or operands). Next, it creates initialization instructions for the uninitialized source operands. It also creates instructions to read the destination operands. Finally, it returns the modified list that contains the initialization operations, modified input operations, and the read operations (in that order).

6.1 Test Generation for Register Read/Write

Algorithm 1 presents the procedure for generating test programs for detecting faults in register read/write functions. The fault model for the register read/write function is described in Section 4.1. For each register in the architecture, the algorithm generates an instruction sequence consisting of a write followed by a read for that register. The function *GenerateUniqueValue* returns unique value for each register based on register name. A test program for register R_i will consist of two assembly instructions: "MOVI R_i, #val_i" and "STORE R_i, R_j, #0". The move-immediate (MOVI) instruction writes val_i in register R_i. The STORE instruction reads the content of R_i and writes it in memory addressed by R_j (offset 0).

```
Algorithm 1: Test Generation for Register Read/Write
Input: Graph model of the architecture G.
Output: Test programs for detecting faults in register read/write.
begin  /*** TestProgramList = {} ***/
    for each register reg in architecture G
        $value_{reg}$ = GenerateUniqueValue(reg);
        writeInst = an instruction that writes $value_{reg}$ in register reg.
        $testprog_{reg}$ = createTestProgram(writeInst)
        TestProgramList = TestProgramList ∪ $testprog_{reg}$;
    endfor
    return TestProgramList.
end
```

6.2 Test Generation for Operation Execution

Algorithm 2 presents the procedure for generating test programs for detecting faults in operation execution. The fault model for the operation execution is described in Section 4.2. The algorithm traverses the behavior graph of the architecture, and generates one test program for each operation graph using *createTestProgram*. For example, a test program for the operation graph with opcode *ADD* in Figure 2 has three operations: two initialization operations ("MOV R3 #333", "MOV R5 #212") followed by the ADD operation ("ADD R2 R3 R5"), followed by the reading of the result ("STORE R2, Rx, #0").

```
Algorithm 2: Test Generation for Operation Execution
Input: Graph model of the architecture G.
Output: Test programs for detecting faults in operation execution.
begin  /*** TestProgramList = {} ***/
    for each operation oper in architecture G
        $testprog_{oper}$ = createTestProgram(oper);
        TestProgramList = TestProgramList ∪ $testprog_{oper}$;
    endfor
    return TestProgramList.
end
```

6.3 Test Generation for Execution Path

Algorithm 3 presents the procedure for generating test programs for detecting faults in execution path. The fault model for the execution path is described in Section 4.3. The algorithm traverses the structure graph of the architecture, and for each pipeline path it generates a group of operations supported by that path. It randomly selects one operation from each operation group. There are two possibilities. If all the edges in the execution path (containing the pipeline path) are activated by the selected operation, the algorithm generates all possible source/destination assignments for that operation. However, if different operations in the operation group activates different set of edges in the execution path, it generates all possible source/destination assignments for each operation in the operation group.

```
Algorithm 3: Test Generation for Execution Path
Input: Graph model of the architecture G.
Output: Test programs for detecting faults in execution path.
begin  /*** TestProgramList = {} ***/
    for each pipeline path path in architecture G
        $opgroup_{path}$ = operations supported in path.
        $exec_{path}$ = path and all data-transfer paths connected to it
        $oper_{path}$ = randomly select an operation from $opgroup_{path}$
        if ($oper_{path}$ activates all edges in $exec_{path}$)  $ops_{path} = oper_{path}$
        else  $ops_{path} = opgroup_{path}$  endif
        for all operations oper in $ops_{path}$
            for all source/destination operands opnd of oper
                for all possible register values val of opnd
                    newOper = assign val to opnd of oper.
                    $testprog_{oper}$ = createTestProgram(newOper).
                    TestProgramList = TestProgramList ∪ $testprog_{oper}$;
                endfor
            endfor
        endfor
    endfor
    return TestProgramList.
end
```

6.4 Test Generation for Pipeline Execution

Algorithm 4 presents the procedure for generating test programs for detecting faults in pipeline execution. The fault model for the pipeline execution is described in Section 4.4. The first loop (L1) traverses the structure graph of the architecture in a bottom-up manner, starting at leaf nodes. The second loop (L2) computes test programs for generating all possible exceptions in each unit using templates. The third loop (L3) computes test programs for creating stall conditions due to data and control hazards in each unit using templates. The fourth loop (L4) creates test programs to generate stall conditions using structural hazards. Finally, the last loop (L5) computes test sequences for multiple exceptions involving more than one units. The *composeTestProgram* function uses ordered[2] n-tuple units and combines their test programs. The function also removes dependencies across test programs to ensure the generation of multiple exceptions during the execution of the combined test program.

```
Algorithm 4: Test Generation for Pipeline Execution
Input: Graph model of the architecture G.
Output: Test programs for detecting faults in pipeline execution.
begin  /*** TestProgramList = {} ***/
        L1: for each unit node unit in architecture G
            L2: for each exception exon possible in unit
                    $template_{exon}$ = template for exception exon
                    $testprog_{unit}$ = createTestProgram($template_{exon}$);
                    TestProgramList = TestProgramList ∪ $testprog_{unit}$;
            endfor
            L3: for each hazard haz in {RAW, WAW, WAR, control}
                    $template_{haz}$ = template for hazard haz
                    if haz is possible in unit
                      $testprog_{unit}$ = createTestProgram($template_{haz}$);
                      TestProgramList = TestProgramList ∪ $testprog_{unit}$;
                    endif
            endfor
            L4: for each parent unit parent of unit
                    $oper_{parent}$ = an operation supported by parent
                    resultOps = createTestProgram($oper_{parent}$);
                    $testprog_{unit}$ = a test program to stall unit (if exists)
                    $testprog_{parent}$ = resultOps ∪ $testprog_{unit}$
                    TestProgramList = TestProgramList ∪ $testprog_{parent}$;
            endfor
        endfor
        L5: for each ordered n-tuple ($unit_1$, $unit_2$, ..., $unit_n$) in graph G
                $prog_1$ = a test program for creating exception in $unit_1$
                .....
                $prog_n$ = a test program for creating exception in $unit_n$
                $testprog_{tuple}$ = composeTestProgram($prog_1$ ∪ ... ∪ $prog_n$);
                TestProgramList = TestProgramList ∪ $testprog_{tuple}$;
        endfor
        return TestProgramList.
end
```

7 A Case Study

We applied our methodology on two pipelined architectures: a VLIW implementation of the DLX architecture [5], and a RISC implementation of the SPARC V8 architecture [19].

7.1 Experimental Setup

We developed our test generation and coverage analysis framework using Verisity's Specman Elite [18]. We captured executable specification of the architectures using Verisity's "e" language. This includes description of 91 instructions for the DLX, and 106 instructions for the SPARC V8 architecture. We refer to these as *specifications*. We implemented a VLIW version of the DLX architecture using Verisity's "e" language. It contains 5 pipeline stages: fetch, decode, execute, memory and writeback. The execute stage has four parallel execution paths: an ALU, a four-stage floating-point adder, a seven-stage multiplier, and a multi-cycle divider. We used the LEON2 processor [20] that is a VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. We refer these models (VLIW DLX and LEON2) as *implementations*.

[2]The unit closer to completion has higher order

Our framework generates test programs in three different ways: random, constrained-random, and our approach. Specman Elite [18] is used to generate both random and constrained-random test programs from the specification. Several constraints are used for constrained-random test generation. For example, to generate test programs for register read/write, we used the highest probability for choosing register-type operations in DLX. Since register-type operations have 3 register operands, the chances of reading/writing registers are higher than immediate type (2 register operands) or branch type (one register operand) operations. The test programs generated by our approach uses the algorithms described in Section 6. To ensure that the generated test programs are executed correctly, our framework applies the test programs on the implementation as well as the specification, and compares the contents of the program counter, registers and memory locations after execution of each test program.

The Specman Elite framework allows definition of various coverage measures that enables us to compute the functional coverage described in Section 5. We defined each entry in the instruction definition (e.g. opcode, destination and sources) as a coverage item in Specman Elite. The coverage for the destination operand gives the measure of which registers are written. Similarly, the coverage of source operands gives the measure of which registers are read. We used a variable for each register to identify a read after a write. Computation of coverage for *operation execution* is done by observing the coverage of the opcode field. The computation of coverage for *execution path* is performed by observing if all the registers are used for computation of all/selected opcodes. This is performed by using cross coverage of instruction fields in Specman Elite that computes every combination of values of the fields. Finally, we compute the coverage for *pipeline execution* by maintaining variables for stalls and exceptions in each unit. The coverage for multiple exceptions is obtained by performing cross coverage of the exception variables (events) that occur simultaneously. Currently, we consider only two simultaneous exceptions.

7.2 Results

In this section, we compare the test programs generated by our approach against the random and constrained-random test programs generated by the Specman Elite. Table 1 shows the comparative results for the DLX architecture. The rows indicate the fault models, and the columns indicate test generation techniques. An entry in the table has two numbers. The first one represents the minimum number of test programs generated by that test generation technique for that fault model. The second number (in parenthesis) represents the functional coverage obtained by the generated test programs for that fault model.

Table 1. Test Programs for Validation of DLX Architecture

Fault Models	Test Generation Techniques		
	Random	Constrained	Our Approach
Register Read/Write	3900 (100%)	750 (100%)	130 (100%)
Operation Execution	437 (100%)	443 (100%)	182 (100%)
Execution Path	12627 (100%)	1126 (100%)	320 (100%)
Pipeline Execution	30000 (25%)	30000 (30%)	626 (100%)

The number 100% implies that the generated test programs covered all the faults in that fault model. For example, the *Random* technique covered all the faults in "*Register Read/Write*" function using 3900 tests. The number of test programs for operation execution are similar for both random and constrained-random approaches. This is because the constraint used in this case (same probability for all opcodes) may be the default option used in random test generation approach.

Table 2. Quality of the Proposed Functional Fault Model

Fault Models	Test Programs	HDL Code Coverage
Register Read/Write	130	85%
Operation Execution	182	91%
Execution Path	320	86%
Pipeline Execution	626	100%

We performed an initial study to evaluate the quality of our functional fault model using existing coverage measures. Table 2 compares our functional coverage against HDL code coverage. The first column indicates the functional fault models. The second column presents the minimum number of test programs generated by our test generation algorithms to cover all the functional faults in the corresponding fault model. The last column presents the code coverage obtained for the DLX implementation [17] using the test programs mentioned in the second column. As expected, our fault model performed well – a small number of test programs generated a high code coverage.

Table 3 shows the comparative results for different test generation approaches for the LEON2 processor. The trend is similar in terms of number of operations and fault coverage for both the DLX and LEON2 architectures. The random and constrained-random approaches obtained 100% functional coverage for the first three fault models using an order of magnitude more test vectors than our approach.

Table 3. Test Programs for Validation of LEON2 Processor

Fault Models	Test Generation Techniques		
	Random	Constrained	Our Approach
Register Read/Write	1746 (100%)	654 (100%)	130 (100%)
Operation Execution	416 (100%)	467 (100%)	212 (100%)
Execution Path	1500 (100%)	475 (100%)	192 (100%)
Pipeline Execution	30000 (40%)	30000 (50%)	248 (100%)

We analyzed the cause for the low fault coverage in *pipeline execution* for the random and constraint-driven test generation approaches. These two approaches covered all the stall scenarios and majority of the single exception faults. However, they could not activate any multiple exception scenarios. Due to bigger pipeline structure (larger set of pipeline interactions) in the VLIW DLX, it has lower fault coverage than the LEON2 architecture in *pipeline execution*. This functional coverage problem will be even more important for today's deeply pipelined embedded processors.

8 Conclusions

Functional verification is widely acknowledged as a major bottleneck in microprocessor design due to lack of a suitable functional coverage estimation technique. This paper presented a functional coverage based test generation technique for pipelined architectures. The methodology made three important contributions. First, a general graph model was developed that can capture the structure and behavior (instruction-set) of a wide variety of pipelined processors. Second, we proposed a functional fault model that is used in defining the functional coverage. Finally, test generation procedures were presented that accept the graph model of the microprocessor as input and generate test programs to detect all the faults in the functional fault model. We are able to measure the goodness of a given set of random test sequences using our functional coverage metric. Our experimental results demonstrate that the required number of test sequences generated by our algorithms to obtain a given fault (functional) coverage is an order of magnitude less than the random or constrained-random test programs.

Our future work includes application of these test programs for functional validation of today's microprocessors. We also plan to perform further comparative studies with our functional coverage metric against existing coverage measures, such as code coverage, FSM coverage and stuck-at coverage.

References

[1] A. Aharon et al. Test program generation for functional verification of PowerPC processors in IBM. *DAC*, pages 279–285, 1995.

[2] Andrew Piziali. *Functional Verification Coverage Measurement and Analysis*. Kluwer Academic Publishers, 2004.

[3] D. Campenhout et al. High-level test generation for design verification of pipelined microprocessors. *DAC*, pages 185–188, 1999.

[4] S. Fine et al. Coverage directed test generation for functional verification using bayesian networks. *DAC*, pages 286–291, 2003.

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

[6] R. Ho et al. Architecture validation for processors. *ISCA*, 1995.

[7] H. Iwashita et al. Automatic test pattern generation for pipelined processors. *ICCAD*, pages 580–583, 1994.

[8] K. Kohno and N. Matsumoto. A new verification methodology for complex pipeline behavior. *DAC*, pages 816–821, 2001.

[9] M. Behm et al. Industrial experience with test generation languages for processor verification. *DAC*, pages 36–40, 2004.

[10] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. *DATE*, pages 182–187, 2004.

[11] J. Miyake et al. Automatic test generation for functional verification of microprocessors. *ATS*, pages 292–297, 1994.

[12] J. Shen et al. Functional verification of the equator MAP1000 microprocessor. *DAC*, pages 169–174, 1999.

[13] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design & Test of Computers*, 18(4):36–45, 2001.

[14] L. Chen et al. A scalable software-based self-test methodology for programmable processors. *DAC*, pages 548–553, 2003.

[15] S. Thatte and J. Abraham. Test generation for microprocessors. *IEEE Transactions on Computers*, C-29(6):429–441, June 1980.

[16] S. Ur and Y. Yadin. Micro architecture coverage directed generation of test programs. *DAC*, pages 175-180, 1999.

[17] http://www.rs.e-technik.tu-darmstadt.de/TUD/res/dlxdocu/SuperscalarDLX.html. *Superscalar DLX Processor*.

[18] Verisity Design, Inc. *http://www.verisity.com*.

[19] http://www.sparc.com/resource.htm#V8. *The SPARC Architecture Manual, Version 8*.

[20] LEON2 Processor. *http://www.gaisler.com/leon.html*.