Functional Verification of Pipelined Processors: A Case Study

Prabhat Mishra
Computer and Info. Science and Engg.
University of Florida
prabhat@cise.ufl.edu

Nikil Dutt
Center for Embedded Computer Systems
University of California, Irvine
dutt@uci.edu

Yaron Kashai
Verisity Design, Inc.
Mountain View, California
yaron@verisity.com

Abstract

Functional verification of pipelined processors is one of the major bottlenecks in current System-on-Chip (SOC) design methodology. A significant bottleneck in the validation of such systems is the lack of a suitable functional coverage metric. This paper presents a test generation and functional coverage estimation framework for pipelined processors using Specman Elite. We have applied this methodology on a VLIW DLX architecture to demonstrate the usefulness of our approach.

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 by applying a combination of random and constrained-random test cases.

Many techniques have been proposed for generation of directed test programs. Aharon et al. [1] have proposed a test program generation methodology for functional verification of PowerPC processors in IBM. Miyake et al. [11] have presented a combined scheme of random test generation and specific sequence generation. A coverage driven test generation technique is presented by Fine et al. [3]. Shen et al. [12] have used the processor to generate tests at run-time by self-modifying code, and performed signature comparison with the one obtained from emulation. These techniques do not consider pipeline behavior for generating test programs.

Ur and Yadin [13] 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. [2] have proposed a test generation 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. [9] have proposed a graph-based functional test program generation technique for pipelined processors using model checking.

Several coverage measures including code coverage, toggle coverage, and FSM coverage are typically used to determine the effectiveness of the generated test programs. However, these measures do not have any direct relationship with the device functionality. For example, none of these determine if all possible interactions of hazards, stalls and exceptions are tested in a processor pipeline.

This paper presents a test generation and functional coverage estimation framework for pipelined processors using Specman Elite. We use an Architecture Description Language (ADL) based specification of pipelined processors as input. The "e" [14] specification is generated from the ADL specification. Specman Elite is used to generate random and constrained-random test programs. We have developed functional coverage estimation techniques to verify the quality of the generated test programs.

The rest of the paper is organized as follows. Section 2 describes our validation methodology followed by a case study in Section 3. Finally, Section 4 concludes the paper.

Proceedings of the fifth International Workshop on Microprocessor Test and Verification (MTV'04)
1550-4093/04 $20.00 © 2004 IEEE

2 Test Generation and Coverage Estimation

Figure 1 shows our test generation and coverage estimation methodology. The first step is to capture the pipelined processor using a specification language. We use the EXPRESSION ADL [4] to describe the structure and behavior of the processor. Our framework enables generation of "e" [14] models for instruction-set architecture (ISA) using the behavioral description of the ADL specification. The ISA specification is used by the Specman Elite to generate test programs. These test programs are applied on the implementation using simulation.

Our framework generates "e" models for the pipelined implementation and uses an "e" simulator. Typically, the implementation is hand-written using Hardware Description Language (HDL) such as VHDL or Verilog. In that case, an HDL simulator is necessary. However, the methodology remains the same. Coverage specification can be generated from the ADL specification or manually written. The coverage specification is used by Specman Elite to generate functional coverage during simulation of the implementation.

The methodology has four important steps: architecture specification, "e" model generation, test generation, and coverage estimation.

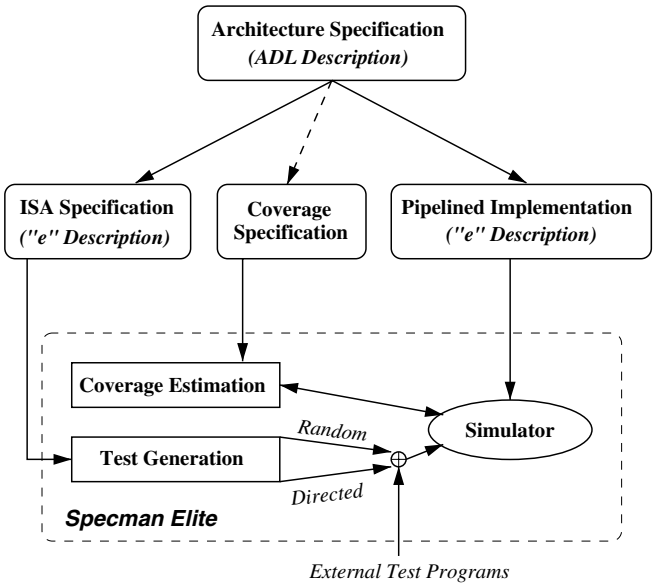

Figure 1. Test Generation and Coverage Estimation

2.1 Architecture Specification

We use the EXPRESSION ADL [4] to describe the structure (components and connectivity) and behavior (instruction-set description) of the processor. It is important to note that any existing ADL can be used that captures both structure and behavior of the processor.

The structure of a processor can be viewed as a graph with the components as nodes and the connectivity as the edges of the graph. 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*. The pipeline edges specify instruction transfer between units via pipeline latches, whereas the data transfer edges specify data transfer between components. Each component has a list of attributes. For example, a functional unit will have information regarding latches, ports, connections, opcodes, timing, capacity etc.

The behavior is organized into operation groups, with each group containing a set of operations having some common characteristics. Each operation is then described in terms of it's opcode, operands, behavior, and instruction format. Each operand is classified either as source or as destination. Furthermore, each operand is associated with a type that describes the type and size of the data it contains. The instruction format describes the fields of the operation in both binary and assembly. For example, the following EXPRESSION code segment describes an ADD operation.

```
(OPCODE add
  (OPERANDS (SRC1 reg) (SRC2 reg/imm) (DEST reg))
  (BEHAVIOR DST = SRC1 + SRC2)
  (FORMAT 000101 dest(25-21) src1(20-16) ...)
  .......
)
```

2.2 Model Generation

Our framework enables generation of "e" models for instruction-set architecture (ISA) using the behavioral description of the ADL specification. The framework also allows generation of pipelined implementation ("e" description) using functional abstraction [10].

2.3 Test Generation

The test programs are generated in three different ways: random, constrained-random, and manual. Spec-

man Elite [14] is used to generate both random and constrained-random test programs from the ISA specification. The external test vectors are manually written for improving the functional coverage.

2.4 Coverage Estimation

Coverage estimation is used to measure the progress of verification effort. It also indicates the quality of the test programs used during simulation. We plan to measure the functional coverage of pipelined processors. We include the following measures in our initial set for coverage estimation:

- **Register Read/Write**: To ensure fault-free execution, all registers should be written and read correctly. A functional fault (bug) in register read/write is covered if the register is written first and read later.
- **Operation Execution**: All operations must execute correctly if there are no faults (bugs). A fault in operation execution is covered if the operation is performed, and the result of the computation is read.
- **Pipeline Execution**: An implementation of a pipeline is faulty if it produces incorrect result due to execution of multiple operations in the pipeline. 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.

3 A Case Study

We applied our methodology on a VLIW implementation of the DLX architecture [5]. Figure 2 shows the simplified version of the VLIW DLX architecture. It has five pipeline stages: fetch, decode, execute, memory (MEM), and writeback. The *execute* stage has four parallel execution paths: integer ALU, 7 stage multiplier (MUL1 - MUL7), four stage floating-point adder (FADD1 - FADD4), and multi-cycle divider (DIV). The oval boxes represent units and rectangular boxes represent storages. The solid lines represent instruction-transfer paths and dotted lines represent data-transfer paths.

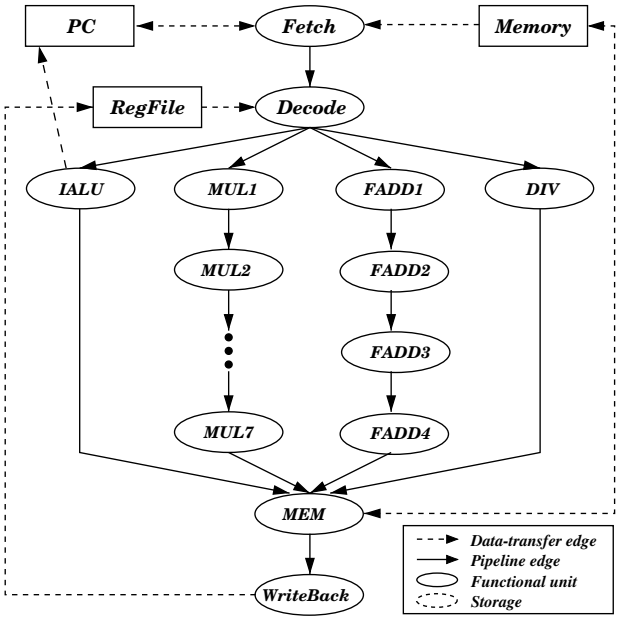

Figure 2. The VLIW DLX architecture

3.1 Experimental Setup

We developed our test generation and coverage analysis framework using Verisity Specman Elite [14]. We captured executable specification of the architectures using Verisity "e" language [14]. This includes description of 91 instructions for the DLX architecture. We refer to it as *specification*. We implemented a VLIW version of the DLX architecture (shown in Figure 2) using "e" language. We refer to it as *implementation*[1].

Our framework uses test programs generated in three different ways: random, constrained-random, and manual. Specman Elite [14] is used to generate both random and constrained-random test programs from the specification. Random test generation uses a variable to decide the number of test programs to generate. 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

[1]The implementation can be written using VHDL/Verilog as well.

register-type operations in DLX. Since register-type (R-type) operations have 3 register operands, the chances of reading/writing registers are higher than immediate type (I-type: 2 register operands) or J-type (no register operand) operations[2].

The external test programs are manually written based on the enumeration of the possible faults. DLX has 32 (32-bit) general purpose registers (GPR) and 32 single-precision (32-bit) floating-point registers. The floating-point registers can be used as 16 double-precision registers. Therefore, total 65 faults possible (64 32-bit registers and one program counter) due to register read/write (65x2=130 test instructions). Similarly, DLX has 91 operations. Therefore, 91 faults possible due to operation execution (91x2=182 test instructions). Finally, let us compute the total number of possible faults in pipeline execution of DLX shown in Figure 2. There are two independent components in this category: stalls and exceptions. A unit can be stalled due to data, control or structural hazards. There are three possible data hazard conditions: RAW, WAW and WAR. Each unit of DLX can be stalled due to structural hazard (17 conditions). The decode unit can be stalled due to data and control hazards (4 conditions) and due to stall of any one of its children (4 conditions). All other units of DLX can be stalled due to stall of its children (16 possibilities). There are 17 single exception and 136 multiple (two) exception scenarios. Multiple exceptions occur when more than one unit are in exception. Currently, we consider scenarios with two simultaneous exceptions. There are total 177 possible faults due to pipeline execution (136x4 + 16x2 + 4x2 + 4x2 + 17x2 = 626 test instructions).

To ensure that the generated test programs are executed correctly by the implementation, 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 as shown in Figure 3.

The Specman Elite framework allows definition of various coverage measures that enables us to compute the functional coverage as described in Section 2.4. 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. Similarly, computation of coverage for operation execution is done by observing the coverage of the opcode field. 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.

[2]DLX supports three types of instruction encoding: I-type, R-type, and J-type

3.2 Results

Table 1 shows the 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 tests generated by that test generation technique for that fault model. The second number (in parenthesis) represents the functional coverage obtained by the generated tests for that fault model.

Table 1. Test Programs for Validation of DLX

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

The number 100% implies that the generated tests 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 tests 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.

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.

Our test generation and coverage estimation framework for the DLX processor is available as an *eShare* (user contributed "e" solutions) in Verisity verification

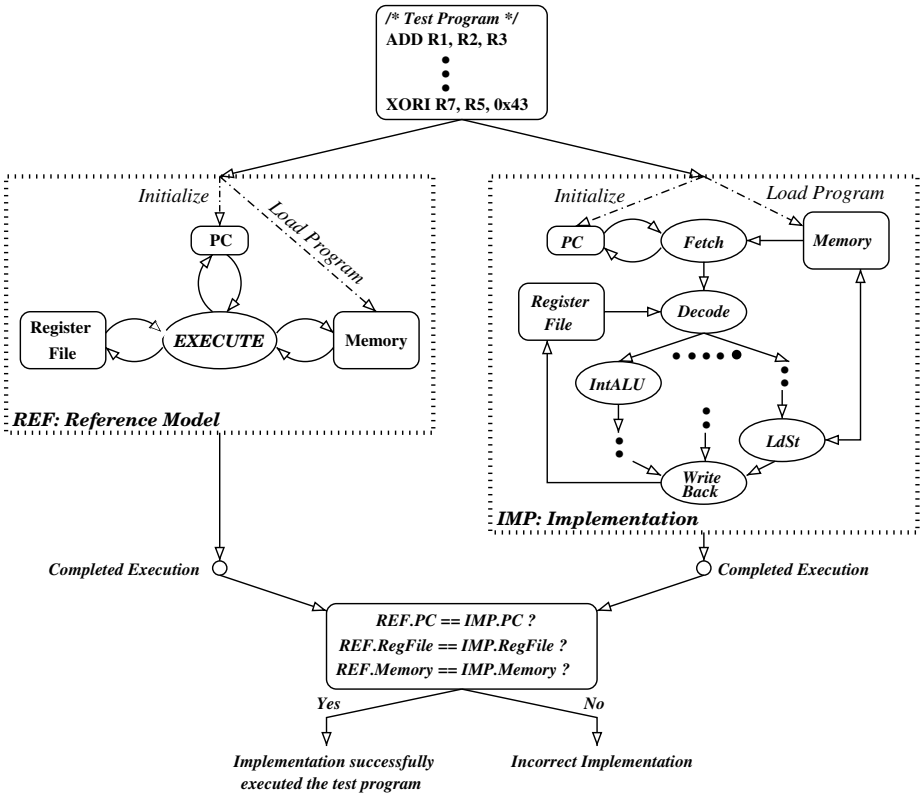

Figure 3. Validation of the Implementation

vault [15]. This includes "e" specifications for both ISA description and pipelined implementation of the DLX architecture. It also includes components for random and constrained-random test generation as well as interface for incorporating external tests during simulation. Finally, it includes components for data and temporal checking, and functional coverage estimation.

4 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 methodology for test generation and coverage estimation of pipelined processors using Specman Elite. We have presented a case study for validating a VLIW implementation of the DLX processor.

Our future work includes development of functional fault models for a wide variety of architectures including RISC, DSP, VLIW, and superscalar. The fault models can be used to define the functional coverage. We also plan to develop functional coverage driven test generation techniques for pipelined processors.

5 Acknowledgments

This work was partially supported by NSF grants CCR-0203813 and CCR-0205712. We would like to thank Verisity university program [14] for giving access to Specman Elite.

References

[1] A. Aharon et al. Test program generation for functional verification of PowerPC processors in IBM. In *Proceedings of Design Automation Conference (DAC)*, pages 279–285, 1995.

[2] D. Campenhout, T. Mudge, and J. Hayes. High-level test generation for design verification of pipelined microprocessors. In *Proceedings of Design Automation Conference (DAC)*, pages 185–188, 1999.

[3] S. Fine and A. Ziv. Coverage directed test generation for functional verification using Bayesian networks. In *Proceedings of Design Automation Conference (DAC)*, pages 286–291, 2003.

[4] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. *DATE*, 1999.

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

[6] R. Ho, C. Yang, M. A. Horowitz, and D. Dill. Architecture validation for processors. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, 1995.

[7] H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose. Automatic test pattern generation for pipelined processors. *ICCAD*, 1994.

[8] K. Kohno and N. Matsumoto. A new verification methodology for complex pipeline behavior. In *Design Automation Conference (DAC)*, 2001.

[9] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. In *Proceedings of Design Automation and Test in Europe (DATE)*, 2004.

[10] P. Mishra, N. Dutt, and A. Nicolau. Functional abstraction driven design space exploration of heterogeneous programmable architectures. In *Proceedings of International Symposium on System Synthesis (ISSS)*, 2001.

[11] J. Miyake, G. Brown, M. Ueda, and T. Nishiyama. Automatic test generation for functional verification of microprocessors. In *Proceedings of Asian Test Symposium (ATS)*, pages 292–297, 1994.

[12] J. Shen, J. Abraham, D. Baker, T. Hurson, M. Kinkade, G. Gervasio, C. Chu, and G. Hu. Functional verification of the equator MAP1000 microprocessor. In *Proceedings of Design Automation Conference (DAC)*, 1999.

[13] S. Ur and Y. Yadin. Micro architecture coverage directed generation of test programs. In *Proceedings of Design Automation Conference (DAC)*, 1999.

[14] Verisity. *http://www.verisity.com*.

[15] *https://www.verificationvault.com*.