Specification-driven Directed Test Generation for Validation of Pipelined Processors

PRABHAT MISHRA
Department of Computer and Information Science and Engineering, University of Florida
NIKIL DUTT
Department of Computer Science, University of California, Irvine

Functional validation is a major bottleneck in pipelined processor design due to the combined effects of increasing design complexity and lack of efficient techniques for directed test generation. Directed test vectors can reduce overall validation effort since shorter tests can obtain the same coverage goal compared to the random tests. This article presents a specification-driven directed test generation methodology. The proposed methodology makes three important contributions. First, a general graph model is developed that can capture the structure and behavior (instruction-set) of a wide variety of pipelined processors. The graph model is generated from the processor specification. Next, we propose a functional fault model that is used to define the functional coverage for pipelined architectures. Finally, we propose two complementary test generation techniques: test generation using model checking, and test generation using template-based procedures. These test generation techniques 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 several orders-of-magnitude reduction in overall validation effort by drastically reducing both test generation time and number of test programs required to achieve a coverage goal.

Categories and Subject Descriptors: D.3.4 [**Software**]: Programming Languages—*Processors*; I.6.7 [**Computing Methodologies**]: Simulation and Modeling—*Simulation Support Systems*

General Terms: Verification, Algorithms

Additional Key Words and Phrases: Model Checking, Test Generation, Functional Validation

1. INTRODUCTION

Functional validation is a major bottleneck in processor design due to the combined effects of increasing design complexity and decreasing time-to-market. Simulation-based validation is the most widely used form of processor verification using test programs that consist of instruction sequences. There are three types of test generation techniques: random, directed, and constrained-random. The directed tests can reduce overall validation effort since shorter tests can obtain the same coverage

Author's address: P. Mishra, Department of Computer and Information Science and Engineering, University of Florida, Gainesville FL 32611, email: prabhat@cise.ufl.edu
N. Dutt, Donald Bren School of Information and Computer Sciences, University of California, Irvine, CA 92697, email: dutt@ics.uci.edu

goal compared to the random tests. Certain heuristics and design abstractions are used to generate directed random testcases. However, due to the bottom-up nature and localized view of these heuristics the generated testcases may not yield a good coverage. As a result, directed test generation is mostly performed by human intervention. Hand-written tests entail laborious and time consuming effort of verification engineers who have deep knowledge of the design under verification. Due to the manual development, it is infeasible to generate all directed tests to achieve a coverage goal. The problem is further aggravated due to the lack of a comprehensive functional coverage metric. Automatic directed test generation based on a comprehensive functional coverage metric is the alternative to address this problem.

We propose a directed test program generation scheme using behavioral knowledge of the pipelined architecture specified in an Architecture Description Language (ADL). The specification is used to generate a graph model of the processor architecture. We define a functional fault model based on the graph coverage. This fault model and the specification are used to generate the directed test programs. We explore two alternatives for directed test generation: test generation using model checking and test generation using template-based procedures. We compare these two methods in terms of their applicability and limitations and propose initial ideas to address some of the practical challenges in applying them on real designs. Model checking based test generation have been proposed in the past to validate software designs [Ammann et al. 1998]. To the best of our knowledge, this technique has not been studied before in the context of top-down validation of pipelined processors. This article makes three important contributions. First, a general graph 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, two complementary test generation techniques 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.

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 article, we present a functional fault model for pipelined processors. The fault model should be applicable to a wide variety 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 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 model. We have developed model-checking based as well as template-based procedures to generate tests that activate the faults in the graph model. We applied our methodology on two pipelined processors: a VLIW implementation of the MIPS architecture [Hennessy and Patterson 2003], and a RISC implementation of the SPARC V8 architecture [Sparc V8]. 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 article is organized as follows. Section 2 presents related work addressing test generation for processor validation. Section 3 – 8 present various steps in our specification-driven test generation methodology followed by a case study in Section 9. Finally, Section 10 concludes the article.

2. RELATED WORK

There has been a lot of research in the area of processor validation using simulation-based techniques as well as formal methods. Various researchers have also proposed hybrid approaches to develop improved validation methodology [Parthasarathy et al. 2002; Bhadra et al. 2004]. In this section we present related work addressing test generation for functional validation of pipelined processors. Traditionally, validation of a microprocessor has been performed by applying a combination of random and directed test programs using simulation techniques. There are many successful test generation frameworks in industry today. For example, Genesys-Pro [Adir et al. 2004], used for functional verification of IBM processors, combines architecture and testing knowledge for efficient test generation. Many techniques have been proposed for generation of directed test programs [Aharon et al. 1995; Fine and Ziv 2003].

Ur and Yadin [1999] have presented a method for generation of assembler test programs that systematically probe the micro-architecture of a PowerPC processor. Iwashita et al. [1994] use an FSM based processor modeling to automatically generate test programs. Campenhout et al. [1999] have proposed a test generation algorithm that integrates high-level treatment of the datapath with low-level treatment of the controller. Ho et al. [1995] have presented a technique for generating test vectors for verifying the corner cases of the design. Recently, Wagner et al. [2005] have presented a Markov model driven random test generator with activity monitors that provides assistance in locating hard-to-find corner-case design bugs and performance problems. 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. [2001]. Piziali [2004] has presented a comprehensive study on functional verification coverage measurement and analysis.

Model checking based techniques have been successfully used in processor verification. Ho et al. [1998] extract controlled token nets from a logic design to perform efficient model checking. Jacobi [2002] used a methodology to verify out-of-order pipelines by combining model checking for the verification of the pipeline control, and theorem proving for the verification of the pipeline functionality. Compositional model checking is used to verify a processor microarchitecture containing most of the features of a modern microprocessor [Jhala and McMillan 2001]. Parthasarathy et al. [2004] have presented a safety property verification framework using sequential SAT and bounded model checking. Model checking based techniques are also used in the context of proving properties or test generation by generating counterexamples. Clarke et al. [1995] have presented an efficient algorithm for generation of counterexamples and witnesses in symbolic model checking. Bjesse et al. [2004] have used counterexample guided abstraction refinement to find complex bugs. This article explores the use of model checking in the context of specification-driven test generation for pipelined processors.

Many researchers have proposed techniques for generation of functional test programs for manufacturing testing of microprocessors ([Krstic et al. 2002], [Thatte and Abraham 1980]). 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. SPECIFICATION-DRIVEN TEST GENERATION

Figure 1 shows our graph based functional test program generation methodology. In our specification-driven test program generation scenario, the designer starts by specifying the processor architecture in an Architecture Description Language (ADL). We use EXPRESSION ADL [Halambi et al. 1999] in our framework. However, our methodology is independent of the ADL. This is due to the fact that all the analysis and test generation techniques, presented in this article, operate on the graph model. Therefore, we can use any ADL in our framework that has information regarding structure and behavior of the processor such as LISA [Zivojnovic at al. 1996], MDES [Gyllenhaal et al. 1996] etc. and thereby enable the generation of the graph-based model of the processor from the ADL specification.

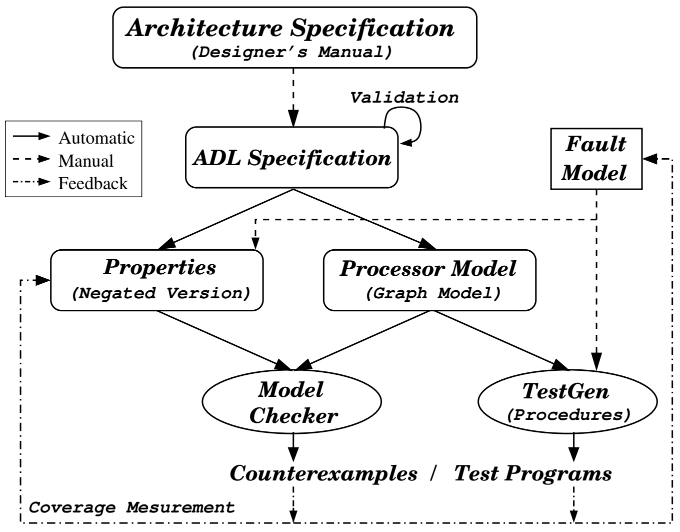

Fig. 1. Test Program Generation Methodology

Test programs are generated from the specification based on the functional fault model using two different test generation techniques: test generation using model checking and test generation using template-based procedures. This methodology has four important steps: ADL specification, processor (graph) model generation, development of fault model, and test generation. Section 4 briefly outlines how to capture a pipelined processor using an ADL specification. Section 5 presents the graph model generation approach followed by the description of the functional fault models in Section 6. Finally, the test generation techniques are presented in Section 7 and Section 8.

4. THE ADL SPECIFICATION

In this section we briefly describe how to specify a pipelined processor using EXPRESSION ADL [Halambi et al. 1999]. As mentioned earlier, any ADL can be used that captures both the structure and the behavior of the processor. The EXPRESSION ADL contains information regarding the structure, behavior and mapping (between structure and behavior) of the processor as shown in Figure 2.

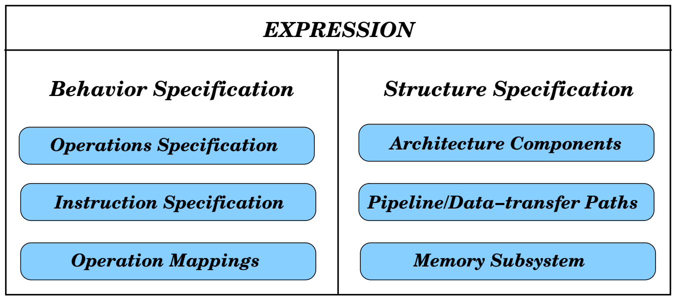

Fig. 2. The EXPRESSION ADL

The structure contains the description of each component and the connectivity between the components. There are four types of components: *units* (e.g., ALUs), *storages* (e.g., register files), *ports*, and *connections* (e.g., buses). A portion of the ADL description of the MIPS processor (shown in Figure 7) is shown below.

```
# Components specification
( ExecUnit IALU
  (capacity 1) (timing (add 1) (sub 1) . . . )
  (opcodes (add sub . . . )) (latches . . . ) . . .
)
. . . . . .
# Pipeline and data-transfer paths
(pipeline Fetch Decode Execute MEM WriteBack)
(Execute IALU MUL FADD DIV)
(MUL MUL1 MUL2 MUL3 MUL4 MUL5 MUL6 MUL7)
(FADD FADD1 FADD2 FADD3 FADD4)
(dtpaths (WriteBack RegFile) (RegFile Decode) . . . )
. . . . . .
```

Each component has a list of attributes. For example, the *IALU* unit has information regarding the number of instructions executed per cycle, timing of each instruction, supported opcodes, and so on. The connectivity is established using pipeline and data-transfer paths. The pipeline edges specify instruction transfer between units via pipeline latches, whereas the data transfer edges specify data transfer between components, typically between units and storages or between two storages. For example, the above ADL specification describes the five-stage pipeline as {*Fetch, Decode, Execute, MEM, Writeback*}. In this particular case, the execute stage has four parallel paths: IALU, MUL, FADD and DIV. Furthermore, each path can contain pipelined or multi-cycle execution units. For example, the FADD path consists of four pipeline stages: FADD1 to FADD4.

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 binary format for the following *add* operation has opcode (0101) field from 26th bit to 29th bit.

```
(OPCODE add
 (OPERANDS (SRC1 reg) (SRC2 reg/imm16) (DST reg))
 (BEHAVIOR DST = SRC1 + SRC2)
 (FORMAT cond(31-30) 0101 dst(25-21) src1(20-16) src2(15-0))
)
```

The mapping functions map components in the structure to operations in the behavior. It defines, for each functional unit, the set of operations supported by that unit (and vice versa). For example, the operation *add* is mapped to the *IALU* unit of the MIPS processor.

5. 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.

5.1 Structure

The structure of an architecture pipeline is modeled as a graph with the components as nodes and the connectivity as 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 3 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*

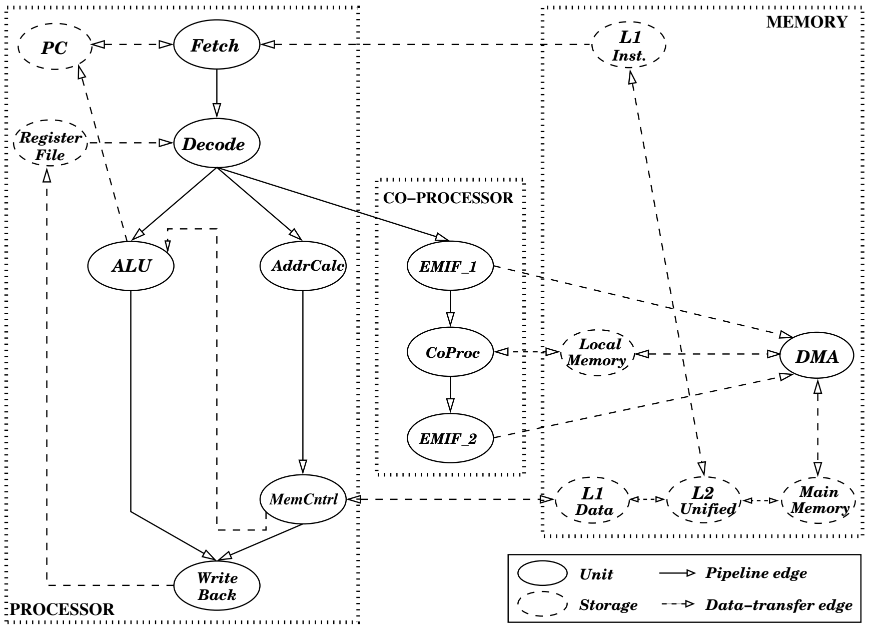

Fig. 3. A Structure Graph of a Simple Architecture

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.

5.2 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 etc.) 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 4 describes a portion of the behavior (consisting of two operation graphs) for the example processor shown in Figure 3.

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 4, 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 4, the solid edges represent operation edges while the dotted edges represent execution

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

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.

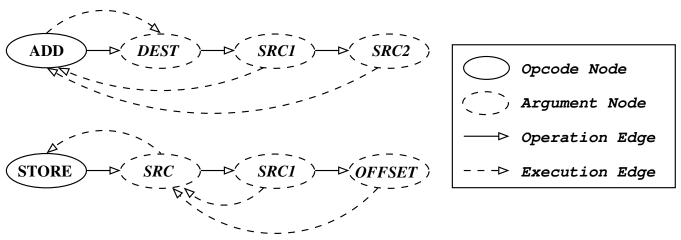

Fig. 4. A Fragment of the Behavior Graph

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 the 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 3 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*.

6. 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*. The fault models described in this section represents various types of faults in a pipelined processor. It is important to note that these fault models are by no means the "golden" model rather it is a representative model which can be refined or modified for improved verification methodology.

The fault models were developed in a way such that we test simple ones first and ensure that the components are working correctly and then we test the complete system consisting of complex components. For example, the *register read/write* fault model ensures that the registers are working correctly. Next, the *operation execution* fault model ensures each operation works correctly assuming that the registers can be read and written correctly. It is possible that a pipelined implementation can have two pipeline paths to execute ALU operations. As long as one of the pipeline paths is working correctly, the *operation execution* fault model will be fine. However, the *execution path* fault model ensures that an operation works correctly on all possible paths. So far we considered only one operation at a time and assumed other operations in the pipeline will not interfere with this operation. The *pipeline execution* fault model considers interactions of multiple operations in the pipeline. Due to the nature of our fault model, there is some redundancy between fault models e.g., *execution path* (considers all possible ways of computing ADD for example) includes scenarios for *operation execution* (one possible way of computing ADD for example). Similarly, *pipeline execution* has some overlap with other fault models.

In this section, 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 5. 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.

6.1 Fault Model for Register Read/Write

The register read/write fault model assumes that each register can be faulty. In other words, reading a register will not return a 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 $\neq V_{R_i}$.

As mentioned earlier, the fault model can be refined or modified for improved verification methodology. For example, consider two different types of errors in register read/write functionality: a particular bit is faulty (always 0, always 1 or not connected to the output) and register selection logic is faulty (e.g., R_4 is selected when R_3 is accessed for both read and write). Clearly, none of these faults can be captured by the current register read/write fault model. The first scenario can be captured by redefining the fault model for any bit can be faulty under register read/write (instead of any register can be faulty). Similarly, the second scenario can be captured by refining the current fault model to define fault in register read, register write as well as in register selection. The impact of such refinement on test generation technique is described in Section 8.

6.2 Fault Model for Operation Execution

The operation execution fault model assumes that each operation could be faulty. In other words, the output of the computation will be different from the expected output after completion of the operation. 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 will not be equal to val_i.

6.3 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., $\exists k \; dest_k \neq val_i^k$

6.4 Fault Model for Pipeline Execution

The previous three fault models consider only one operation at a time. In other words, the other operations in the pipeline does not interact or influence the flow of this operation. The pipeline execution fault model defines the possible errors in the presence of interactions between multiple operation in the pipeline. 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, erroneous data forwarding 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 $ExceptionSet = \cup_{\forall u} ES_u \cup MESS$. Similarly, other interactions can be defined. Let us define all possible pipeline interactions as PIs. 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., $\exists k \; dest_k \neq val_i^k$.

7. TEST GENERATION USING MODEL CHECKING

Algorithm 1 shows the three major steps in test generation using model checking. We use SMV model checker [SMV] in our framework. The first step produces one property (negated version) for each fault. The second step generates the SMV model of the processor architecture. The final step applies each property and the processor model to the model checker to generate the counterexample. The counterexample is analyzed to generate the test program consisting of instruction sequences.

```
Algorithm 1: Test Generation using Model Checking
Inputs: 1. Graph Model of the architecture ($G$).
        2. Functional fault model ($F$).
Output: Test programs for detecting all the faults in the fault model.
begin   /*** $PropertyList = \{\}$ ***/
    Step 1: for each fault $f$ in the fault model $F$
                $prop_f$ = GenerateProperty($f$); /*SMV version*/
                $PropertyList = PropertyList \cup prop_f$;
            endfor
    Step 2: $design$ = GenerateSMVmodel($G$)
    Step 3: $TestProgramList = \{\}$
            for each property $p$ in the $PropertyList$
                $test_p$ = ApplyModelChecking($p$, $design$);
                $TestProgramList = TestProgramList \cup test_p$;
            endfor
    return $TestProgramList$.
end
```

For example, to generate a testcase for assigning a value *5* to a register R_7, the property states that "R_7 *!= 5*". The model checker produces a counterexample which is converted to a test program. The conversion is straightforward in our framework since we model the design at the cycle-accurate level and instructions are modeled as a structure consisting of opcode, source and destination operands. As a result, the counter-example consists of several instructions at different clock cycles which can be translated into an actual test by simple text analysis. Based on the coverage report additional properties can be added or the fault model can be modified. Section 9.1 presents a case study for test generation using model checking. The remainder of this section is organized as follows. Section 7.1 describes how to generate SMV models from the ADL specification. Section 7.2 describes the procedure for property generation based on fault models. Finally, Section 7.3 describes test generation based on model checking using the generated SMV models and properties.

7.1 SMV Model Generation for Pipelined Processors

Generation of SMV models from ADL specification is similar to the existing ADL-driven approaches for cycle-accurate simulator generation [Mishra et al. 2001] and synthesizable RTL generation [Mishra et al. 2004]. The only difference is that the SMV generation approach uses the component library described in SMV, whereas the simulator generation approach uses the component library described in C/C++.

Similarly, the RTL generation approach uses the component library described using VHDL/Verilog. An alternative way to generate SMV models is to generate Verilog models from the ADL specification and then convert the Verilog description to SMV using *vl2smv* translator [SMV]. These two approaches are shown in Figure 5.

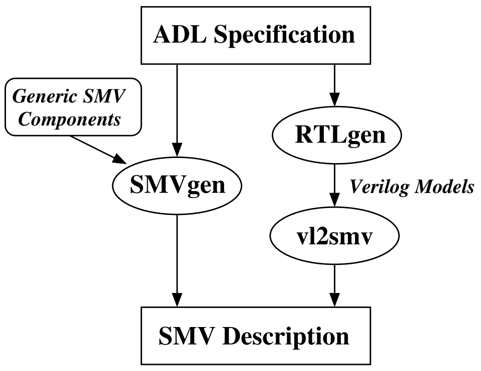

Fig. 5. Generation of SMV Models

The basic idea of SMV model generation (SMVgen) is to develop a library of generic components that can be composed to construct a pipelined processor. We have developed a set of generic components and sub-components (described using SMV) based on the functional abstraction defined by Mishra et al. [2001]. Section 9.1 shows an example *Fetch* unit (generic component) in SMV library. The development of generic SMV models is a one-time activity and independent of the architecture. The SMV description is generated by composing the abstraction primitives based on the information available in the ADL specification. The SMV generation process consists of three steps. First, the ADL specification is read to gather all the necessary details. Next, the functionality of each component is composed using the generic functions (components) and sub-functions. Finally, the structure of the processor is composed using the structural details. In the remainder of this section we briefly describe how to generate three major components of the processor: instruction decoder, datapath and controller, using the generic SMV models.

A generic instruction decoder uses information regarding individual instruction format and opcode mapping for each functional unit to decode a given instruction. The instruction format information is available in the operation description. The decoder extracts information regarding opcode, operands etc. from input instruction using the instruction format. The mapping section of the EXPRESSION ADL has the information regarding the mapping of opcodes to the functional units. The decoder uses this information to perform/initiate necessary functions (e.g., operand read) and decide where to send the instruction.

The implementation of datapath consists of two parts. First, compose each component in the structure. Second, instantiate components (e.g., fetch, decode, ALU, LdSt, writeback, branch, caches, register files, memories etc.) and establish connectivity using the structural information available in the ADL. To compose each component in the structure we use the information available in the ADL regarding the functionality of the component and its parameters. For example, to compose an execution unit, it is necessary to instantiate all the opcode functionalities (e.g, ADD, SUB etc. for an ALU) supported by that execution unit. Also, if the execution unit is supposed to read the operands, appropriate number of operand read functionalities need to be instantiated unless the same read functionality can be shared. Similarly, if this execution unit is supposed to write the data back to register file, the functionality for writing the result needs to be instantiated.

The controller is implemented in two parts. First, it generates a centralized controller (using generic controller function with appropriate parameters) that maintains the information regarding each functional unit, such as busy, stalled etc. It also computes hazard information based on the list of instructions currently in the pipeline. Based on these bits and the information available in the ADL, it stalls/flushes necessary units in the pipeline. Second, a local controller is maintained at each functional unit in the pipeline. This local controller generates certain control signals and sets necessary bits based on the input instruction. For example, the local controller in an execution unit will activate the add operation if the opcode is *add*, or it will set the busy bit in case of a multi-cycle operation.

7.2 Generation of Properties

The properties are generated based on the fault models. For example, if we adopt a *node fault* model for a pipelined processor, we need to generate one property for each node (functional unit or storage) in the graph model of the pipelined processor. Section 9.1 shows several examples of faults (functional errors) and corresponding temporal logic properties.

The generated properties are expressed in linear temporal logic (LTL) [Clarke et al. 1999] where each property consists of temporal operators (G, F, X, U) and Boolean connectives ($\wedge$, $\vee$, $\neg$, and $\rightarrow$). We generate a property for each fault in the fault model. Each fault may be related to one (local errors) or more (pipeline interactions) functions units. Such interactions can be converted in the form of a property such as $F(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$ that combines activities p_i over n modules using logical AND operator. The atomic proposition p_i is a functional activity at a node (module) i such as operation execution, stall, exception or NOP. The property is true when all the p_is (i = 1 to n) hold at some time step. Since we are interested in counterexample generation, we need to generate the negation of the property first. The negation of the properties can be expressed as:

$$\begin{aligned} \neg X(p) = X(\neg p), \neg G(p) = F(\neg p) \\ \neg F(p) = G(\neg p), \neg pUq = pR\neg q \end{aligned} \qquad (1)$$

For example, the negation of $F(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$, *interaction fault*, can be described as $G(\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_n)$ whose counterexamples will satisfy the original property.

7.3 Test Generation

The model checking based test generation (Algorithm 1) is a promising approach for automated generation of directed micro-architectural tests to exercise various intricate interactions and corner cases. Since the complete processor is used during model checking, this approach is limited by the capacity restrictions of the model checking tool. As a result, this approach is not suitable for today's pipelined processors since the time and memory requirements can be prohibitively large in many test generation scenarios. In fact, test generation may not be possible in various instances due to state space explosion. We propose a test generation approach using decomposition of the processor model and properties to make the ADL-driven test generation applicable in practice.

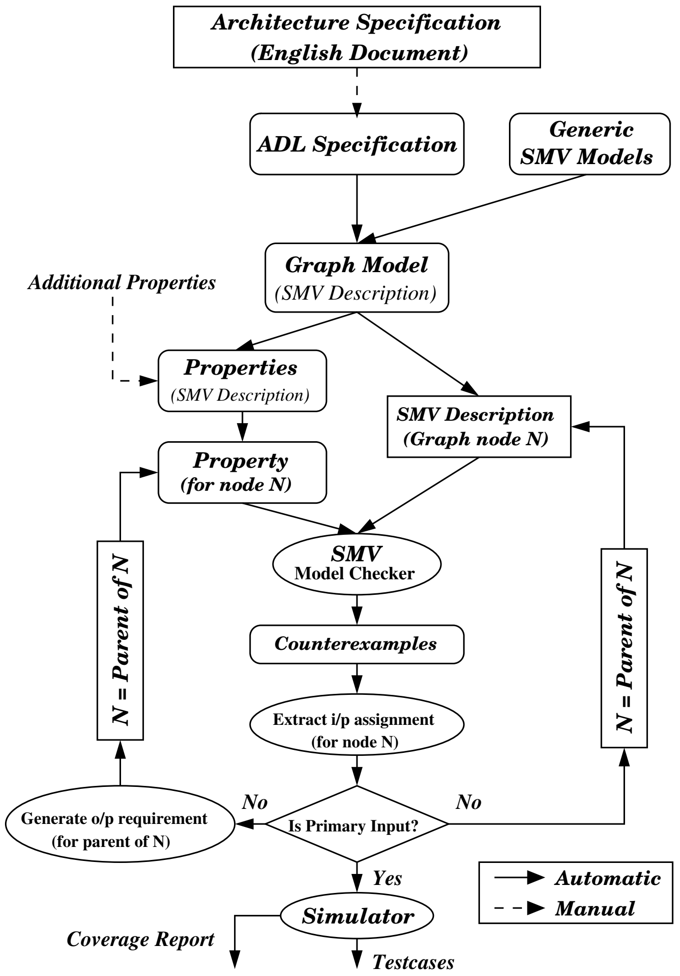

Fig. 6. Test Generation using Decompositional Model Checking

Figure 6 shows our test generation framework using decompositional model checking. The basic idea is to break one processor level property into multiple module level properties and apply them to the respective modules. In case the property was applied to an internal module, the generated counterexample is used to extract the output requirements for the parent module(s). This iteration continues until the primary input assignments (e.g., assignment to register file or instruction memory) are obtained. These primary input assignments are converted into test programs consisting of instruction sequences. Since, the model checker is applied only at the module level, this approach can handle larger designs and reduces the test generation time. It is important to note that various constraints has to be captured in an efficient manner to generate a useful test. We model global input and environmental constraints as part of the design. We model local input constraints as part of the property during decomposition to ensure that model checker always generates correct partial counterexamples.

```
Algorithm 2: Test Program Generation
Inputs: ADL specification of the pipelined processor
Outputs: Test programs to verify the pipeline behavior.
Begin
  Generate graph model of the architecture.
  Generate properties based on the graph coverage
  for each property prop for graph node n
      inputs = φ
      while (inputs != primary_inputs)
            Apply prop on node n using SMV model checker
            inputs = Find i/p requirements for n from counterexample
            if inputs are not primary_inputs
               Extract output requirements for parent of node n
               prop = modify prop with new output requirements
               n = parent of node n
            endif
      endwhile
      Convert primary input assignments to a test program
      Generate the expected output using a simulator.
  endfor
  return the test programs
End
```

Algorithm 2 describes the major steps in Figure 6. A property *prop* is applied to a module corresponding to node n in the graph model. As mentioned earlier, the framework generates the negation of the properties that we want to verify. The model checker produces a counterexample for the property *prop*. The counter example is analyzed to find the input requirements for the node n. If these inputs are not the primary inputs of the processor, the output requirements for the parent node of n is computed. The property is modified based on the output requirements and applied to the parent node. This iteration continues until primary input assignments are obtained. These primary input assignments are converted into test programs (instruction sequences) by putting random values in the un-assigned inputs. Section 9.1 presents a test generation example using the decomposition of processor model and properties.

Test generation using module level decompositions is very useful but may not be able to generate tests where module level decompositions are not possible. We have developed novel design and property decomposition techniques [Koo and Mishra

2006a] as well as SAT-based bounded model checking techniques [Koo and Mishra 2006b] to address the state space explosion problem in test generation. Our initial study using Freescale e500 processor [e500 2005] shows promising results [Koo et al. 2006] in applying decompositional model checking based test generation on industrial microprocessors.

There are various important issues that need to be considered during test generation using decompositions. For example, when a test can be obtained by many possible decompositions, it is difficult to determine the most profitable decomposition. It is equally difficult to guarantee that a given decomposition is sufficient to generate a test since a local property may not generate a counterexample. In general, when there are multiple decompositions possible, and test generation did not get counterexample for a local scenario, the algorithm needs to consider other decomposition scenarios. This may lead to multiple iterations and thereby increased test generation time. Another major problem in decompositional model checking based test generation is how to merge the local counterexamples since it may introduce conflicts in parent nodes. We have addressed these issues in [Koo and Mishra 2006a].

The test generation requires the model of the design as well as a property. Depending on the complexity of the design and the test generation scenario the property can be very complex. Designer may consider to write the testcase directly instead of writing a property and then take all the trouble of performing decompositional model checking. There are various reasons why writing a testcase by hand is not a good idea. When a fault model is available, our framework will be able to generate the properties automatically for all the faults in the fault model. Of course, there will be certain corner cases when designers have to write few properties by hand (on top of the ones generated automatically). It is important to note that writing a complex property may require few minutes to hours. However, writing a complex testcase may require days or even weeks. Most importantly, the hand-written testcase may not even activate the fault since it needs to consider so many constraints over a period of time.

8. TEST GENERATION USING TEMPLATE-BASED PROCEDURES

In this section, we present test generation procedures for detecting faults covered by the fault models presented in Section 6. Different architectures have specific instructions to observe the contents of registers and memories. In this article, 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. Procedure 1 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).

```
Procedure 1: createTestProgram
Input: An operation list operList.
Output: Modified operation list with initializations.
begin
        resOperations = {};
        for each operation oper in operList
            if there are unspecified fields in oper
              assign appropriate opcode/operands;
            endif
            for each source src of oper
                if (src is a register or memory location) then
                  initOper: initialize src with appropriate value;
                  resOperations = resOperations $\cup$ initOper;
                endif
            endfor
            resOperations = resOperations $\cup$ oper;
            for each destination dest of oper
                if (dest is not a source for another operation)
                  readDest: create an instruction to read dest;
                  resOperations = resOperations $\cup$ readDest;
                endif
            endfor
        endfor
        return resOperations.
end
```

Consider an input list with one operation *"ADD dest/reg R1 src2/imm"*. The operation has two unspecified fields: dest and src2. The *createTestProgram* function assigns a register to *dest* field and an immediate value to *src2* field. It also creates an initialization operation for the source *R1*. The modified list consists of three operations: *MOVI R1, 0x5* followed by *ADD R3 R5 0x23* and *STORE R3, Rx, 0x0*. We use 'STORE' instruction for observability of registers. If an architecture has a specific instruction for this purpose, that specific instruction should be used instead of the store instruction.

8.1 Test Generation for Register Read/Write

Algorithm 3 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 6.1. 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.

As mentioned in Section 6 that the test generation techniques need to be modified if the fault model is refined. Consider the two refinements of the register read/write fault model: any bit can be faulty under register read write (instead of any register can be faulty), and fault can be in register selection (not just register read and write). To address the first refinement the test generation technique needs to generate tests that write and read different sets of values including all 1's and all 0's. The second refinement can be addressed by modifying the test generation technique that assigns a specific value (say all 1's) to a register and another value (say all 0's) to all other registers, and generates read instructions in a particular order. This process needs to continue for all registers to ensure that there are no errors in register selection/decoding.

```
Algorithm 3: 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.
        /* The read instruction is created inside createTestProgram() */
        $testprog_{reg}$ = createTestProgram(writeInst)
        TestProgramList = TestProgramList ∪ $testprog_{reg}$;
    endfor
    return TestProgramList.
end
```

THEOREM 8.1. *The test sequence generated using Algorithm 3 is capable of detecting any detectable fault in the register read/write fault model.*

PROOF. Algorithm 3 generates one test program for each register in the architecture. A test program consists of two instructions - a write followed by a read. Each register is written with a specific value. If there is a fault in register read/write function, the value read would be different that the written value. □

8.2 Test Generation for Operation Execution

Algorithm 4 presents the procedure for generating test programs for detecting faults in operation execution. The fault model for the operation execution is described in Section 6.2. 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 4 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").

THEOREM 8.2. *The test sequence generated using Algorithm 4 is capable of detecting any detectable fault in the operation execution fault model.*

PROOF. Algorithm 4 generates one test program for each operation in the architecture. If there is a fault in operation execution, the computed result would be different than the expected output. □

```
**Algorithm 4**: *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($oper$);
            $TestProgramList = TestProgramList \cup testprog_{oper}$;
        endfor
        return $TestProgramList$.
end
```

8.3 Test Generation for Execution Path

Algorithm 5 presents the procedure for generating test programs for detecting faults in execution path. The fault model for the execution path is described in Section 6.3.

```
**Algorithm 5**: *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).
                        $TestProgramList = TestProgramList \cup testprog_{oper}$;
                    endfor
                endfor
            endfor
        endfor
        return $TestProgramList$.
end
```

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.

THEOREM 8.3. *The test sequence generated using Algorithm 5 is capable of detecting any detectable fault in the execution path fault model.*

PROOF. The proof is by contradiction. The only way a detectable fault will be missed if an pipeline or data-transfer edge is not activated (used) by the generated test programs. Let us assume, an edge e_{pp} is not activated by any operation. If the e_{pp} is not part of (connected to) any pipeline path, the fault is not detectable. Let us further assume, e_{pp} is part of pipeline path *pp*. If the pipeline path e_{pp} does not support any operations, the fault is not detectable. If it does support operations, Algorithm 5 will generate operation sequences that exercises this pipeline path and all the data-transfer paths connected to it. Since, the edge e_{pp} is connected to pipeline path *pp*, it is activated. □

8.4 Test Generation for Pipeline Execution

Algorithm 6 presents the procedure for generating test programs for detecting faults in pipeline execution. The fault model for the pipeline execution is described in Section 6.4. 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.

An important requirement in test generation is the availability of templates. In general, it is not possible to create template for one instance and apply for other instances. Consider the following template that can be used to generate an exception in the DIV (division) unit. This template consists of only one instruction where both destination and first source operand is register. However, the second source operand is zero. As a result, this will cause a divide by zero exception.

```
DIV <reg>  <reg>  #0
```

Clearly, the above template is not helpful in creating an exception in a *Decode* unit. There are many possible templates for creating an exception in *Decode* unit for example, we can employ an instruction sequence (VLIW instruction) where a particular slot has an instruction of incorrect type (e.g., ADD operation in multiply slot for the MIPS processor) to cause an exception of illegal instruction. Similarly, there are various ways of stalling *Decode* unit, we can employ the following in-

[2]The unit closer to completion has higher order

struction sequence (template) which stalls the unit due to read-after-write (RAW) hazard involving R1 as both source and destination of two consecutive instructions.

```
<opcode>   R1  <reg> <reg/imm>
<opcode> <reg>  R1   <reg/imm>
```

This assumes that the *Decode* unit does not have any reservation station, therefore, RAW will cause the unit to stall. In the presence of an instruction buffer (reservation station), we need to use an instruction sequence (template) that will fill the buffer and stall the unit.

```
Algorithm 6: 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 (or specific event) $exon$ possible in $unit$
                $template_{exon}$ = template for exception $exon$
                $testprog_{unit}$ = createTestProgram($template_{exon}$);
                $TestProgramList = TestProgramList \cup testprog_{unit}$;
        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($template_{haz}$);
                    $TestProgramList = TestProgramList \cup testprog_{unit}$;
                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)
                $testprog_{parent} = resultOps \cup testprog_{unit}$
                $TestProgramList = TestProgramList \cup testprog_{parent}$;
        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($prog_1 \cup ... \cup prog_n$);
            $TestProgramList = TestProgramList \cup testprog_{tuple}$;
    endfor
    return $TestProgramList$.
end
```

THEOREM 8.4. *The test sequence generated using Algorithm 6 is capable of detecting any detectable fault in the pipeline execution.*

PROOF. Algorithm 6 generates test programs for all possible interactions during pipeline execution. The first for loop (L1) generates all possible hazard and exception scenarios for each functional unit in the pipeline. The test programs for creating all possible exceptions in each node are generated by the second for loop (L2). The third for loop (L3) generates test programs for creating all possible data and control hazards in each node. Similarly, the fourth loop (L4) generates tests for creating all possible structural hazards in a node. Finally, the last loop (L5) generates test programs for creating all possible multiple exception scenarios in the pipeline. □

9. A CASE STUDY

This section describes case studies of applying different test generation techniques (model checking and template-based procedures) using different pipelined architectures. First, we present results using model checking based test generation approach. Next, we present the results using template-based test generation procedures.

9.1 Test Generation using Model Checking

We applied our methodology on a single-issue MIPS [Hennessy and Patterson 2003] architecture. Figure 7 shows the simplified version of the VLIW MIPS 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 dashed boxes represent storages. The solid lines represent instruction-transfer paths and dotted lines represent data-transfer paths.

We used SMV model checker [SMV] for test generation. Therefore, the properties as well as the processor model are described using SMV language. The SMV model of the processor is generated from the ADL specification using functional abstraction [Mishra et al. 2001]. The basic idea is to develop a library consisting of a set of generic functions and sub-functions (one-time activity and independent of architecture) and compose them based on the ADL specification. For example, a simplified version of the instruction fetch unit in the library is shown below.

```
module Fetch (PC, InstMemory, operation)
{
   input PC : integer;
   input InstMemory : memory;
   output operation : opType;

   init(operation.opcode) := NOP;
   next(operation) := InstMemory[PC];
}
```

The SMV description of the MIPS architecture is generated automatically from the ADL specification using the component library. The SMV description of the MIPS architecture has 431 lines of code using the pipeline and cycle-accurate com-

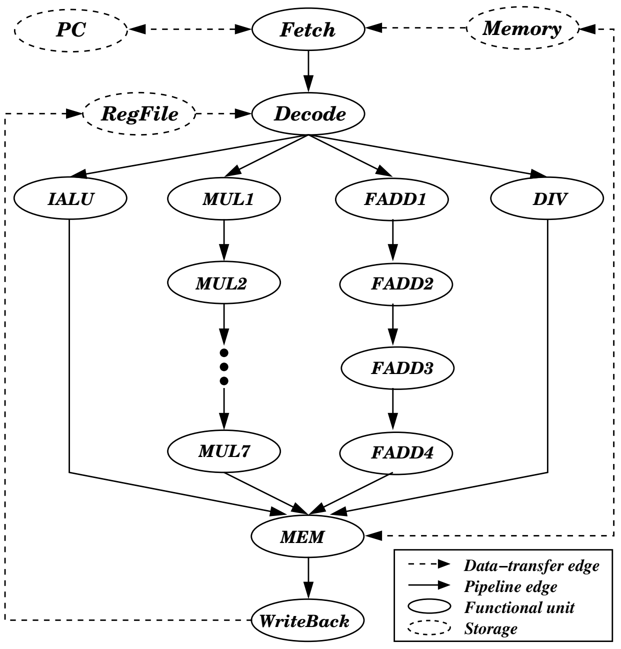

Fig. 7. The VLIW MIPS architecture

ponents from the library [Mishra and Dutt 2002]. The properties are generated based on the fault model described in in Section 6. For example, the following property is used to generate a testcase for stalling the decode unit.

```
hazard: assert G(ID._stall = 0);
```

Our model checking based framework took two seconds on a 333 MHz Sun UltraSPARC-II with 128M RAM. The instruction sequence is shown below. The read-after-write hazard sets the *_stall* bit in this scenario. The *ADD* operation is supported by integer ALU (EX) unit. The decode unit (ID) will be stalled in cycle 4 in this case.

```
Fetch Cycle Opcode Dest Src1 Src2
----------- ------ ---- ---- ----
  1           NOP
  2           ADD    R3, R1, R2
  3           ADD    R4, R3, R2
```

Test generation using model checking is fast for such simple properties. However, the test generation time is very long for complex properties as demonstrated using the following example.

Example 1: *Consider a fragment of the MIPS pipeline containing three internal registers of the division unit (DIV) as shown in Figure 8. Our goal is to initialize two registers A_{in} and B_{in} with values 2 and 3 respectively at clock cycle 9.*

The two internal input registers for DIV unit are A_{in} and B_{in}. The internal output register for DIV unit is C_{out}. The input instruction is *divInst* and the output is *result*. In this particular scenario, A_{in} and B_{in} receive data from the first and second source operands of the input instruction (*divInst*) i.e., $A_{in} = divInst.src1$ and $B_{in} = divInst.src2$; C_{out} returns the result of the division i.e., $C_{out} = A_{in} \div B_{in}$; finally the output is fed from C_{out} i.e., $result = C_{out}$.

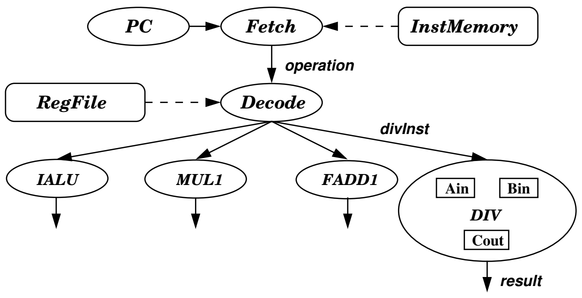

Fig. 8. A fragment of the MIPS architecture

The following property generates the instruction sequence to initialize A_{in} and B_{in} with values 2 and 3 respectively at clock cycle 9. The property is written using SMV language [SMV]. Informally speaking, it implies that if current clock cycle is 8, in the next cycle *DIV.Ain* should not be 2 or *DIV.Bin* should not be 3.

```
assert G((cycle = 8) -> X((DIV.Ain ~= 2) | (DIV.Bin ~= 3)));
```

If this property is applied on the complete description of the MIPS processor (using Algorithm 1) to generate the required test program, it will take 375.98 seconds and 1928568 BDD nodes on a 333 MHz Sun UltraSPARC-II with 128M RAM. However, the test generation time is drastically reduced if Algorithm 2 is used that performs design and property decompositions, as demonstrated below.

We modify this global property to make it applicable at module level (as shown below) and apply to the division unit (*DIV*) using SMV.

```
assert G((cycle=8) -> X((Ain ~= 2) | (Bin ~= 3)));
```

The next step is to analyze the counterexample produced by SMV to extract the input requirements for the division unit. For example, in this case the input requirements are simple: *divInst.src1 = 2* and *divInst.src2 = 3*. These input requirements are used to generate the expected output assignments for the decode unit (parent of the division unit). Also, the cycle count requirement is modified for the decode unit. The modified property (shown below) is applied to the decode unit.

```
assert G((cycle = 7) -> X((divInst.src1 ~= 2) | (divInst.src2 ~= 3)));
```

The counterexample is analyzed to extract the input requirements for the decode unit. The decode has two inputs: *operation* and *RegFile*. For example, in this case the input requirements are: *operation.opcode = DIV, operation.src1 = 1, operation.src2 = 2, RegFile[1] = 2*, and *RegFile[2]=3*. This indicates that the *operation* should be a division operation with *src1* as R1 and *src2* as R2. It also implies that the register file should have the values 2 and 3 at locations 1 and 2 respectively. There are two tasks to be done here. First, initialize a register file location with a specific value at a given a clock cycle *t*. It is done using a *move-immediate* instruction fetched at *(t-5)*. In this case, the *move-immediate* operations should be done at clock cycle 2 and 3 to make the data available at cycle 8. The second task is to convert the remaining input requirements as the expected outputs for the fetch unit (parent of the decode). The modified property (shown below) is applied to the fetch unit (*Fetch*).

```
assert G((cycle=6) -> X((operation.opcode ~= DIV) |
  (operation.src1 ~= 1) | (operation.src2 ~= 2)));
```

The counterexample is analyzed to extract the input requirements for the fetch unit. The fetch unit has two inputs: *PC* and instruction memory. The expected value for PC is 5 and InstMemory[5] has instruction: $DIV\ R_x\ R_1\ R_2$. These are primary inputs for the processor. The final test program, shown below, is constructed by putting random values in the unspecified fields:

```
Fetch Cycle Opcode Dest Src1 Src2      Comments
----------- ------ ---- ---- ----   --------------
 1           NOP                    R0 is always 0
 2           ADDI  R1,  R0,  #2     R1 = 2
 3           ADDI  R2,  R0,  #3     R2 = 3
 4           NOP
 5           NOP
 6           NOP
 7           DIV   R3,  R1,  R2
```

The system took less than a second to come up with the counterexample on a 333 MHz Sun UltraSPARC-II with 128M RAM. This time includes the time taken by SMV in verifying three module level properties. It also includes the time taken by our system in traversing the graph and generating the new properties with input/output computations using counterexamples. The total number of BDD nodes allocated is 5600. Clearly, Algorithm 2 reduced the test generation time and the required BDD size by an order of magnitude compared to Algorithm 1.

9.2 Test Generation using Template-based Procedures

We applied our methodology on two pipelined architectures: a VLIW implementation of the MIPS architecture [Hennessy and Patterson 2003] (shown in Figure 7)), and a RISC implementation of the SPARC V8 architecture [Sparc V8].

9.2.1 *Experimental Setup.* Figure 9 shows our test generation and coverage analysis framework using Specman Elite [Cadence]. We captured executable specification of the architectures using "e" language. This includes description of 91 instructions for the MIPS, and 106 instructions for the SPARC V8 architecture. We refer to these as *specifications*. We implemented a VLIW version of the MIPS architecture (shown in Figure 7) using "e" language. We used the LEON2 processor [LEON2] that is a VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. We refer these models (VLIW MIPS and LEON2) as *implementations*.

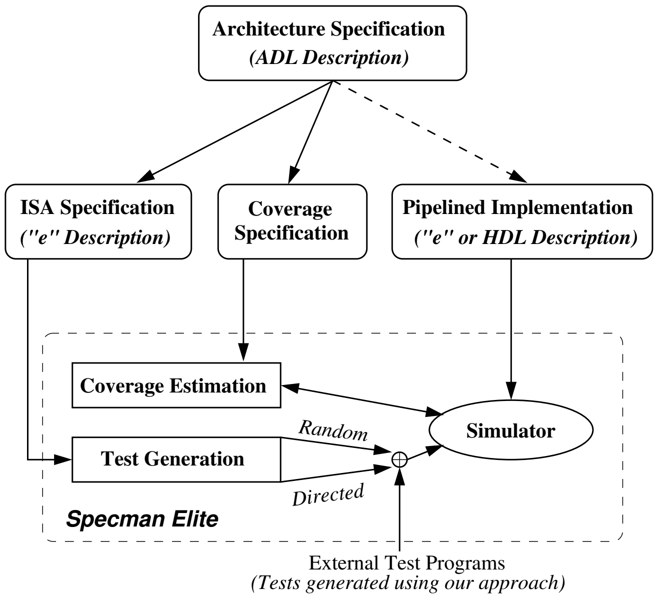

Fig. 9. Test Generation and Coverage Measurement

Our framework generates test programs in three different ways: random, constrained-random, and our approach. Specman Elite [Cadence] 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 MIPS. 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 8.

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. We have performed micro-architectural validation of the pipelined processor by converting the assembly test sequences generated by our method into the testbenches for RTL simulation of the implementation. The simulator shows how instructions go through the pipeline stages on a cycle-by-cycle basis as well as whether the stored results in register files and memory are correct or not. Capturing when and which instructions move from one stage to the next ensures that the generated tests exercise the target micro-architectural artifacts.

The Specman Elite framework allows definition of various coverage measures that enables us to compute the functional coverage described in Section 6. 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.

9.2.2 *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 I shows the comparative results for the MIPS 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 I. Test Programs for Validation of MIPS 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.

It is clear that the test generated by our approach requires a significantly less number of tests to obtain the 100% coverage. However, it is also important to compare the test generation time as well as test application time for all the approaches. The average time to generate one testcase using *random* or *constrained random* approaches are comparable to our template based test generation approach

– typically ranges in the order of seconds. Our model checking based test generation approach requires several minutes on an average for test generation (10-20X slower). However, since the number of tests required to obtain a coverage goal using our approach is drastically less (100-1000X), the overall time for test generation is either comparable or less by orders or magnitude. It is important to note that it may not be always possible to achieve 100% functional coverage using random and constrained tests irrespective of how many such tests are generated. The average time to run a test (test application time) is comparable (in the order of minutes) irrespective of how they are generated since the length of each test is typically 10-100 instructions. However, since number of tests generated by our approach is drastically less, the overall validation time will be reduced by several orders of magnitude compared to random and directed random tests.

Table II 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 MIPS 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. 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 MIPS, 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.

Table II. 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%)

In the remainder of this section we evaluate the quality of our fault model in two ways. First, we compare our fault model and corresponding coverage with existing code coverage metric. Next, we apply the tests generated using our fault model on 40 buggy implementations of DLX and Alpha processors. These buggy implementations were developed by Wagner et al. [2005] at the University of Michigan - Ann Arbor.

Comparison with Code Coverage. We performed an initial study to evaluate the quality of our functional fault model using existing coverage measures. Table III compares our functional coverage against HDL code coverage (statement 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 MIPS implementation [SuperscalarDLX] 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. We have also measured the code coverage using the random and constrained tests and observed that an order of magnitude more tests are required to obtain 100% code coverage.

Table III. 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%

Detection of Bugs in Faulty Implementations. To further evaluate the effectiveness of our fault model, we have applied the tests generated using our fault model on 40 buggy implementations [Wagner et al. 2005] of DLX and Alpha processors. Section 11 describes all the bugs in the DLX and Alpha pipeline and how the tests generated using our fault model detect these bugs. Figure 10 shows the result of our analysis for DLX bugs. In the figure '*v*' denotes that the bug (in X-axis) is activated by the test generated using the corresponding fault model (in Y-axis). The '*ex*' symbol indicates that this bug can be captured by extending the corresponding fault model. For example, the bug 22 requires extension of current register read/write fault model by considering a sequence of two writes followed by a read (instead of one write and then read). The description of the required extensions and other details are available in Section 11.

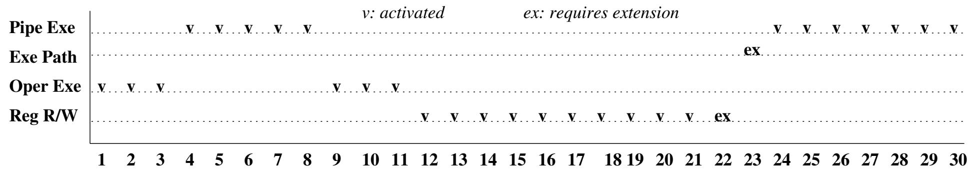

Fig. 10. Coverage of DLX Processor Bugs

Figure 11 shows the similar analysis results for Alpha bugs. Majority of these bugs consist of value specific corner case scenarios. To activate these corner case scenarios require appropriate extensions to our current fault model. For example, the bug 7 requires extension of the current operation execution fault model with specific value assignments. Section 11 describes the required extensions to activate these faults.

We have applied the tests generated by *random* and constrained-random approaches to detect these bugs and obtained results similar to Wagner et al. [2005].

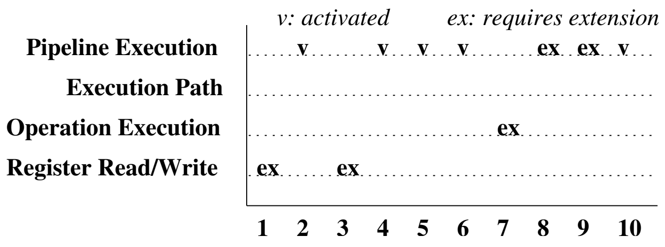

Fig. 11. Coverage of Alpha Processor Bugs

The random and constrained-random approaches requires several thousands or millions of instructions to activate these tests. In other words, the directed test programs generated by our approach can cover majority of these faults by using several orders of magnitude less number of test programs compared to the random and constrained-random approaches.

10. CONCLUSIONS

Functional verification is widely acknowledged as a major bottleneck in microprocessor design due to lack of a comprehensive functional coverage metric and the attendant task of coverage-driven directed test generation technique. This article presented a specification-driven directed test generation methodology based on a functional fault model of pipelined processors. This methodology has three important steps: architecture specification using an ADL, development of a graph model and corresponding functional fault model, and coverage-driven test generation. We have described two test generation approaches: test generation using model checking, and test generation using template-based procedures. These two approaches are complementary and have their own merits and demerits.

Model checking based approach assumes that the processor model and the negated version of the properties are applied using a model checker to generate counterexamples. However, due to the state space explosion problem this technique cannot be applied on large designs or in the presence of complex properties. In certain instances when the test is trying to activate a fault within first few cycles, SAT-based bounded model checking is useful [Koo and Mishra 2006b]. However, in general it is very difficult to determine the bound upfront. Alternatively, various design and property decomposition techniques can be applied to reduce the state space during model checking. However, this may introduce additional challenges in terms of merging the local counterexamples to obtain the global counterexample (final test program) [Koo and Mishra 2006a].

The test generation procedures (without model checking) are very efficient since it can handle large designs. Our experimental results demonstrated 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. However, these procedures assume the availability of necessary templates. In general, it is not possible to create template for one instance and apply for other instances. For example, if we have a template to create an exception in a *Fetch* unit, this template may not be helpful in creating an exception in a *Decode* unit. In such circumstances, we can use model checker to create the necessary template. However, use of a model checker will introduce all the difficulties described above including state space explosion problem.

Based on our experience, test generation procedures are effective when the fault model is simple (templates can be generated and reused easily) and design is large. The model checking based test generation is efficient when the fault model is complicated but the design is small. We plan to study various design and property decomposition techniques for test generation in the context of complex fault model (properties) as well as industry-strength pipelined processors.

11. APPENDIX

This appendix describes various buggy implementations of DLX and Alpha pipelines developed by Wagner et al. [2005] at the University of Michigan - Ann Arbor. These faulty implementations are used in Section 9 to demonstrate how our fault model captured these bugs.

11.1 Buggy DLX Implementations

This section describes the 30 bugs in the DLX pipeline and how they are captured by the tests generated using our fault model. The bugs are listed based on the increasing complexity to detect them (determined by the developers [Wagner et al. 2005]). The bugs 1, 2, 3, 9, 10 an 11 are captured by the tests generated using *operation execution* fault model. The bugs 12 to 21 are detected by the tests generated using *register read/write* fault model. The bugs 4, 5, 6, 7, 8, 24 to 30 are detected by the tests generated using *pipeline execution* fault model. The remaining bugs (bug 22 and 23) can be captured only by extending the current fault model. For example, the bug 22 requires extension of current register read/write fault model by considering a sequence of two writes followed by a read (instead of one write and then read). Similarly. the bug 23 requires extension of execution path fault model by considering specific values for source and destination operands.

(1) SLL shifts the wrong way.
(2) When instruction is ADD, ADDI, ADDU, ADDIU operation performed is logic AND instead of addition.
(3) SLTIU selects the wrong ALU operation. (SLT is selected instead of SLTU).
(4) Whenever there is a JAL instruction, the stage 4 bypass is always activated for the following instruction, because of wrong test in assigning rsr31.
(5) When bypassing from the MEM stage back to the ID stage with an ALU immediate instruction, the data is grabbed from stage EX, instead of MEM.
(6) Missing pipeline stall for the case where Load instruction is followed by BNE/BEQ and the loaded value is used for the comparison in the branch instruction.
(7) In classifying instruction at the MEM stage, the check for the instruction being a JAL and then assigns link_reg to the stage.
(8) When classifying the instruction at the ID stage, it checks for a specific group of special instructions. The bug is that the test checks for the type of instruction at the EX stage instead of the type of special instruction at the ID stage.

(9) The PC value is selected wrong for some branch instructions: with BEQ, BNE, BLEZ, BGTZ and BLTZ the PC is simply added to the relative destination but not jumped.
(10) BGTZ jumps for any value less than 0, but not for values greater than 0.
(11) Missing code to write back instruction result from ALU.
(12) Does not write to register 31 instead of register 0.
(13) Reads to register 20 as RS (source read port) return old RS.
(14) Reads of RT (target read port) return old RT if (RT[0] == RS[4]).
(15) Forwarding through RS (source read port) will fail.
(16) For RTs (target read port) 0...31 reads from 0..15 and 0..15
(17) RT (target read port) reads lower 30 bits only.
(18) Reads to RS are correct only if write to register that is not r0 is in progress.
(19) Write to register 5 after read from r5 as RS does not go through.
(20) Mux always statement of RS (source read port)
(21) If RT was 7 then writes negative value to memory.
(22) If write to r7 is followed by instr with RT=r7 write to r14 occurs.
(23) Load result shifted 1 left if ID_RS=SMDR_4==9.
(24) Load data is corrupted in case of load to the same address as last store.
(25) If LW and ADDI have same immediate, ADDI doesn't sign extend the immediate.
(26) BNE followed by BEQ to the same offset will fail.
(27) ADD to address followed by a store to the same address and any instruction with source of the same address in source fails.
(28) Bypass from MEM stage instead of EX.
(29) addi.rs after a store.rt gets store address.
(30) ADD after a branch with the same RS is not squashed if branch is taken.

11.2 Buggy Alpha Implementations

This section describes the 10 bugs in the Alpha pipeline and how they are captured by the tests generated using our fault model. The bugs are listed based on the increasing complexity to detect them (determined by the developers [Wagner et al. 2005]). The bugs 2, 4 to 6, and 10 are detected by the tests generated using *pipeline execution*. The remaining bugs are captured by extending the current fault model. For example, the bug 1 and 3 requires extension of current register read/write fault model by considering specific value assignments. The bug 7 requires extension of the current operation execution fault model with specific value assignments. The bugs 8 and 9 requires extension of the current pipeline execution fault model with specific value assignments. Clearly, some of the bugs are value-specific corner cases and requires value specific extensions. Any fault model that allow consideration of specific values are not good fault models since it may lead to generation of potentially infinite number of testcases.

(1) Write to zero-reg succeeds if rdb_idx = 5.
(2) Load following store to the same address produces bad result.

(3) Internal forwarding in rda.
(4) Forwarding through zero reg on rb.
(5) Forwarding from wb instead of mem when both match.
(6) Unconditional branch following conditional branch with the same immediate will not be taken.
(7) When wb_rd == rda and first 3 bits of them are 101, last bit of rda is flipped.
(8) If mem and ex is dependent on ra and ra=20, rb is forwarded from writeback.
(9) Conditional branch with negative immediate after a store to positive address is not taken.
(10) Squash if mem_wb ra == id_ex rd and instruction in id ex is not any branch.

ACKNOWLEDGMENTS

We would like to thank Ilya Wagner, Prof. Valeria Bertacco and Prof. Todd Austin of the University of Michigan at Ann Arbor for giving us the opportunity to use the test suite of various DLX and Alpha pipeline implementations.

REFERENCES

Adir, A., Almog, E., Fournier, L., Marcus, E., Rimon, M., Vinov, M., and Ziv, A. 2004. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Design & Test of Computers 21,* 2 (Mar-Apr), 84–93.

Aharon, A., Goodman, D., Levinger, M., Lichtenstein, Y., Malka, Y., Metzger, C., Molcho, M., Shurek, G. 1995. Test program generation for functional verification of PowerPC processors in IBM. In *Proceedings of Design Automation Conference (DAC).* 279–285.

Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N., and Nicolau, A. 1999. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of Design Automation and Test in Europe (DATE).* 485–490.

Krstic, A., Lai, W., Cheng, K-T., Chen, L., and Dey, S.. 2002. Embedded software-based self-test for programmable core-based designs. *IEEE Design & Test of Computers 19,* 4, 18–27.

Piziali, A. 2004. *Functional Verification Coverage Measurement and Analysis.* Kluwer Academic Publishers.

Jacobi, C. 2002. Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving. In *Proceedings of Computer Aided Verification (CAV)*, E. Brinksma and K. Larsen, Ed. LNCS, vol. 2404. Springer-Verlag, 309–323.

Cadence Design Systems, Inc. *http://www.cadence.com.*

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

Clarke, E., Grumberg, O., and Peled, D. 1999. *Model Checking.* MIT Press, Cambridge, MA.

Clarke, E., Grumberg, O., McMillan, K., and Zhao, X. 1995. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of Design Automation Conference (DAC).* 427–432.

Parthasarathy, G., Iyer, M., Cheng, K-T., and Wang, L. 2004. Safety property verification using sequential SAT and bounded model checking. *IEEE Design & Test of Computers 21,* 2, 132–143.

Parthasarathy, G., Iyer, M., Feng, T., Wang, L-C., Cheng, K-T., and Abadir, M. 2002. Combining atpg and symbolic simulation for efficient validation of array systems. In *Proceedings of International Test Conference (ITC).* 203–212.

Iwashita, H., Kowatari, S., Nakata, T., Hirose, F. 1994. Automatic test program generation for pipelined processors. In *Proceedings of International Conference on Computer-Aided Design (ICCAD).* 580–583.

KOO, H., AND MISHRA, P. 2006a. Functional test generation using property decompositions for validation of pipelined processors. In *Proceedings of Design Automation and Test in Europe (DATE)*. 1240–1245.

KOO, H., AND MISHRA, P. 2006b. Test generation using SAT-based bounded model checking for validation of pipelined processors. In *Proceedings of ACM Great Lakes Sympoisum on VLSI (GLSVLSI)*.

KOO, H-M., MISHRA, P., BHADRA, J., AND ABADIR, M. 2006. Directed micro-architectural test generation for an industrial processor: A case study. In *Proceedings of Microprocessor Test and Verification (MTV)*.

Symbolic Model Verifier. http://www.cs.cmu.edu/˜modelcheck.

http://www.freescale.com/files/32bit/doc/ref_manual/e500CORERM.pdf. PowerPC^{TM} e500 Core Family Reference Manual 2005.

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

WAGNER, I., BERTACCO, V., AUSTIN, T. 2005. Stresstest: An automatic approach to test generation via activity monitors. In *Proceedings of Design Automation Conference (DAC)*. 783–788.

BHADRA, J., KRISHNAMURTHY, N., AND ABADIR, M. 2004. Enhanced equivalence checking: Towards a solidarity of functional verification and manufacturing test generation. *IEEE Design & Test of Computers 21,* 6 (Nov-Dec), 494–502.

GYLLENHAAL, J., RAU, B., HWU, W. 1996. HMDES version 2.0 specification. Tech. Rep. IMPACT-96-3, IMPACT Research Group, Univ. of Illinois, Urbana. IL.

HENNESSY, J., AND PATTERSON, D. 2003. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers.

LEON2 Processor. *http://www.gaisler.com/products/leon2/leon.html*.

AMMANN, P., BLACK, P., AND MAJURSKI, W. 1998. Using model checking to generate tests from specifications. In *Proceedings of International Conference on Formal Engineering Methods (ICFEM)*. 46–54.

BJESSE, P., AND KUKULA, J. 2004. Using counter example guided abstraction refinement to find complex bugs. In *Proceedings of Design Automation and Test in Europe (DATE)*. 156–161.

HO, P., ISLES, A., AND KAM, T. 1998. Formal verification of pipeline control using controlled token nets and abstract interpretation. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*. 529–536.

MISHRA, P., AND DUTT, N. 2002. Architecture description language driven functional test program generation for microprocessors using smv. Tech. Rep. CECS 02-26, University of California, Irvine.

MISHRA, P., AND DUTT, N. 2004. Graph-based functional test program generation for pipelined processors. In *Proceedings of Design Automation and Test in Europe (DATE)*. 182–187.

MISHRA, P., KEJARIWAL, A., AND DUTT, N. 2004. Synthesis-driven exploration of pipleined embedded processors. In *Proceedings of International Conference on VLSI Design*. 921–926.

MISHRA, P., AND DUTT, N. 2005. Functional coverage driven test generation for validation of pipelined processors. In *Proceedings of Design Automation and Test in Europe (DATE)*. 678–683.

MISHRA, P., DUTT, N., AND NICOLAU, A. 2001. Functional abstraction driven design space exploration of heterogeneous programmable architectures. In *Proceedings of International Symposium on System Synthesis (ISSS)*. 256–261.

HO. R., YANG, C., HOROWITZ, M., AND DILL, D. 1995. Architecture validation for processors. In *Proceedings of International Symposium on Computer Architecture (ISCA)*.

JHALA, R., AND MCMILLAN, K. 2001. Microarchitecture verification by compositional model checking. In *Proceedings of Computer Aided Verification (CAV)*, G. Berry et al., Ed. LNCS, vol. 2102. Springer-Verlag, 396–410.

FINE, S., AND ZIV, A. 2003. Coverage directed test generation for functional verification using bayesian networks. In *Proceedings of Design Automation Conference (DAC)*. 286–291.

TASIRAN, S., AND KEUTZER, K. 2001. Coverage metrics for functional validation of hardware designs. *IEEE Design & Test of Computers 18,* 4 (Jul-Aug), 36–45.

Thatte, S., and Abraham, J. 1980. Test generation for microprocessors. *IEEE Transactions on Computers C-29,* 6, 429–441.

Ur, S., and Yadin, Y. 1999. Micro architecture coverage directed generation of test programs. In *Proceedings of Design Automation Conference (DAC)*. 175–180.

Zivojnovic, V., Pees, S., and Meyr, H. 1996. LISA - machine description language and generic machine model for HW/SW co-design. In *IEEE Workshop on VLSI Signal Processing*. 127–136.

http://www.rs.e-technik.tu-darmstadt.de/TUD/res/dlxdocu/SuperscalarDLX.html. A Superscalar Version of the DLX Processor.

Received September 2007; revised February 2008; accepted March 2008