Graph-based Functional Test Program Generation for Pipelined Processors[*]

Prabhat Mishra
pmishra@cecs.uci.edu

Nikil Dutt
dutt@cecs.uci.edu

Architectures and Compilers for Embedded Systems (ACES)
Center for Embedded Computer Systems, University of California, Irvine, CA 92697, USA

Abstract

Functional verification is widely acknowledged as a major bottleneck in microprocessor design. While early work on specification driven functional test program generation has proposed several promising ideas, many challenges remain in applying them to realistic embedded processors. We present a graph coverage based functional test program generation approach for pipelined processors. The proposed methodology makes three important contributions. First, it automatically generates the graph model of the pipelined processor from the specification using functional abstraction. Second, it generates functional test programs based on the coverage of the pipeline behavior. Finally, the test generation time is drastically reduced due to the use of module level property checking. We applied this methodology on the DLX processor to demonstrate the usefulness of our approach.

1 Introduction

As embedded systems continue to face increasingly higher performance requirements, deeply pipelined processor architectures are being employed to meet desired system performance. Validation of such programmable processors is one of the most complex and expensive tasks in the current Systems-on-Chip design methodology. Simulation is the most widely used form of microprocessor verification: millions of cycles are spent during simulation using a combination of random and directed test cases in traditional design flow. 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. The problem is further aggravated due to the lack of a comprehensive functional coverage metric.

Specification driven test generation has been introduced as a promising top-down validation technique for pipelined processors [12]. The processor is specified using an Architecture Description Language (ADL). The SMV (Symbolic Model Verifier) [8] description of the processor is generated from the ADL specification of the architecture. Specific properties are applied to the processor model using SMV model checker. For example, to generate a testcase to stall the decode unit, the property states that *the decode unit is not stalled*. The model checker produces a counter example that stalls the decode unit. The generated counterexample is converted into a test program consisting of processor instructions. Since, the complete processor is modeled using SMV, this approach is limited by the capacity restrictions of the tool. As a result, it is not possible to model detailed description of the processor and generate test programs. Furthermore, the test generation time is long.

To make the ADL driven test generation applicable to realistic embedded processors, each of the above steps must be automated using efficient techniques. First, the processor model generation from the specification needs to be automated. Second, there is a need for a comprehensive functional coverage metric that can be used to automatically generate test programs. Finally, an efficient test generation technique is needed that can model complex designs and can enable fast generation of functional test programs.

We propose a graph coverage based test generation technique for functional verification of pipelined processors. The contribution of this paper is a methodology that solves the three problems mentioned above. First, we present a technique for automatic generation of processor model from the ADL specification using functional abstraction. Second, we define functional coverage of the pipeline behavior in terms of pipeline graph coverage. The pipeline graph is generated from the ADL specification of the processor. Each node in the graph corresponds to a functional unit (module) or storage component in the processor. The behavior of each node is described using SMV [8] language. An edge in the graph represents instruction (or data) transfer between the nodes. Finally, we present a test program generation algorithm that traverses the pipeline graph to generate test programs based on the coverage metric. The algorithm breaks one processor level property into multiple module level properties and applies them. Since, the SMV is applied only at the module level, this approach can handle larger designs. It also drastically reduces the test generation time.

The rest of the paper is organized as follows. Section 2

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

presents related work addressing verification of pipelined processors. Section 3 describes our functional test program generation methodology followed by a case study in Section 4. Section 5 concludes the paper.

2 Related Work

Several approaches for formal or semi-formal verification of pipelined processors have been developed in the past. Theorem proving techniques, for example, have been successfully adapted to verify pipelined processors [14]. Burch and Dill presented a technique for formally verifying pipelined processor control circuitry [2]. The technique has been extended to handle more complex pipelined architectures by several researchers [16]. Ho et al. [7] extract controlled token nets from a logic design to perform efficient model checking. Jhala et al. [10] used compositional model checking to verify a modern microprocessor.

Traditionally, validation of a microprocessor has been performed by applying a combination of random and directed test programs using simulation techniques. Many techniques have been proposed for generation of directed test programs. Aharon et al. [1] have proposed a test program generation methodology for functional verification of PowerPC processors in IBM. Shen et al. [15] have used the processor to generate tests at run-time by self-modifying code, and performed signature comparison with the one obtained from emulation. These techniques does not consider pipeline behavior for generating test programs.

Ur and Yadin [18] presented a method for generation of assembler test programs that systematically probe the microarchitecture of a PowerPC processor. Iwashita et al. [9] use a FSM based processor modeling to automatically generate test programs. Campenhout et al. [3] have proposed a test generation algorithm that integrates high-level treatment of the datapath with low-level treatment of the controller. These techniques does not provide a comprehensive metric to measure the coverage of the pipeline interactions.

Many researchers have proposed techniques for generation of functional test programs for manufacturing testing of microprocessors ([4], [11], [17]). These techniques use stuck-at fault coverage to demonstrate the quality of the generated tests. The applicability of these test programs are not shown for functional validation of microprocessors.

3 Functional Test Program 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 [5] in our framework. Our methodology is independent of the ADL. As a result, we can use any ADL that captures both the structure and the behavior of the processor.

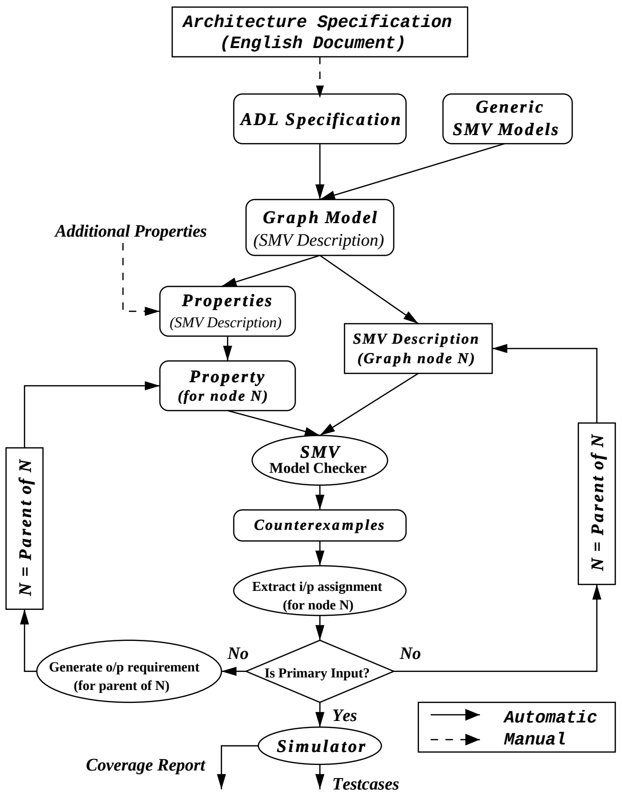
Architecture Specification (English Document)
ADL Specification
Generic SMV Models
Graph Model (SMV Description)
Additional Properties
Properties (SMV Description)
SMV Description (Graph node N)
Property (for node N)
N = Parent of N
SMV Model Checker
Counterexamples
Extract i/p assignment (for node N)
No
Is Primary Input?
Generate o/p requirement (for parent of N)
Yes
Simulator
Coverage Report
Testcases
Automatic
Manual

Figure 1. Test Program Generation Methodology

The graph model of the processor is generated from the ADL specification. The properties are generated based on the graph coverage metric discussed in Section 3.4. The properties are applied at the module level using SMV model checker. The counter examples are analyzed to generate test programs at the processor level. We apply these test programs to the simulator of the processor to ensure that the coverage criteria is met. If necessary, additional properties can be added manually.

Our technique drastically reduces the time and space required for generating the test programs by applying properties at the module level and composing the responses in sequence by traversing the pipeline graph.

Algorithm 1 presents our specification driven test generation procedure. A property *prop* is applied to a module corresponding to node *n* in the graph model. The framework actually generates the negation of the properties that we want to verify. For example, to generate a testcase for assigning a value *5* to a register *R7*, the property states that "*R7 != 5*". The SMV 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 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.

In the remainder of this section we describe each of the steps in detail. First, we describe the EXPRESSION ADL followed by a brief description of the functional abstraction technique. Next, we present the graph model generation technique using functional abstraction. Finally, we define the graph coverage metric that is used for test generation.

```
Algorithm 1: 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
```

3.1 The ADL Specification

The EXPRESSION ADL [5] contains information regarding the structure, behavior and mapping (between structure and behavior) of the processor. 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). Each component has a list of attributes. For example, a functional unit will have information regarding latches, ports, connections, supported opcodes, execution timing, and capacity. The connectivity is established using pipeline and data-transfer paths. The behavior contains the description of operations in terms of its opcode, operands, behavior and instruction format. Finally, the mapping functions map operations in the behavior to components in the structure. For example, an operation *add* is mapped to *ALU* unit in a typical processor.

3.2 The Functional Abstraction

The functional abstraction technique was first introduced by Mishra et al. [13] for generating simulation models from the ADL specification. The notion of functional abstraction comes from a simple observation: different architectures may use the same functional unit (e.g., fetch) with different parameters, the same functionality (e.g., operand read) may be used in different functional unit, or may have new architectural features. The first difference can be eliminated by defining generic functions with appropriate parameters. The second difference can be eliminated by defining generic sub-functions, which can be used by different architectures at different points. The last one is difficult to alleviate since it is new, unless this new functionality can be composed of existing sub-functions (e.g., *multiply-accumulate* operation by combining *multiply* and *add* operations). They defined the necessary generic functions, sub-functions and computational environment to capture a wide variety of processor and memory features.

The structure of each functional unit is captured using parameterized functions. For example, a fetch unit functionality contains several parameters, such as number of operations read per cycle, reservation station size, branch prediction scheme etc. Figure 2 shows a specific example of a fetch unit described using sub-functions. Each sub-function is defined using appropriate parameters. For example, *ReadInstMemory* reads *n* operations from instruction cache using current PC address (returned by *ReadPC*) and writes them to the reservation station. The notion of generic sub-function allows the flexibility of specifying the system in finer detail. It also allows reuse of the components.

```
FetchUnit ( # of read/cycle, res-station size, ....)
{
   address = ReadPC();
   instructions = ReadInstMemory(address, n);
   WriteToReservationStation(instructions, n);
   outInst = ReadFromReservationStation(m);
   WriteLatch(decode_latch, outInst);

   pred = QueryPredictor(address);
   if pred {
      nextPC = QueryBTB(address);
      SetPC(nextPC);
   } else
      IncrementPC(x);
}
```

Figure 2. A Fetch Unit Example

The behavior of a generic processor is captured through the definition of opcodes. Each opcode is defined as a function with a generic set of parameters, which performs the intended functionality. Similarly, they defined generic functions and sub-functions for memory modules, controller, interrupts, exceptions, DMA, and co-processor. The detailed description of generic abstractions for all of the microarchitectural components can be found in [13].

3.3 Graph Model Generation

The structure of a processor pipeline is modeled as a graph $G = (V, E)$. V denotes two types of components in the processor: *functional units* and *storages*. E consists of two types of edges: *pipeline edges* and *data-transfer edges*. A pipeline edge transfers an instruction from a parent unit to a child unit.

A data-transfer edge is used to transfer data between components. Figure 3 shows the graph model of the DLX processor. The oval (unit) and rectangular (storage) boxes represent nodes. The solid (pipeline) and dotted (data-transfer) lines represent edges.

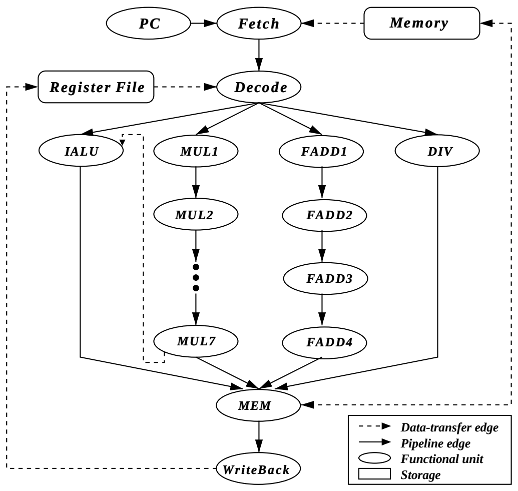

Figure 3. The Graph Model of the DLX Architecture

Each node of the graph contains information regarding input/output edges, list of supported operations and their timing. Each node also contains the SMV description of its behavior. We have implemented all the generic functions and sub-functions described in Section 3.2 using SMV language. Our framework generates SMV description of each node (functional unit/storage) by composing functional abstraction primitives. For example, a simplified version of the SMV description of the fetch unit (*Fetch*) 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];
}
```

3.4 Coverage Directed Test Generation

Measuring progress is an important task that enables the designer to decide when to end the verification effort. Several coverage measures are commonly used, such as code coverage, toggle coverage and fault coverage. Unfortunately, these measures do not have any direct relation to the functionality of the device. For example, none of these determine if all possible interactions of hazards, stalls and exceptions are tested in a processor pipeline. We propose a coverage metric based on functional coverage of the pipeline. We define all possible interactions between opcodes (instructions) and pipeline stages (paths) through graph coverage.

We define graph coverage as graph node coverage and graph edge coverage. A node in the graph is called covered if it has been in all of the four states: active, stalled, exception and flushed. A node is *active* when it is executing an instruction. A node can be *stalled* due to structural or data hazards. A node can be in *exception* state if it generates an exception while executing an instruction. It is possible to have multiple exception scenarios and stall conditions for a node. However, our current node coverage requires only one scenario in each case. A node is in *flushed* state if an instruction in the node is flushed due to the occurrence of an exception in any of its children nodes.

Similarly, an edge in the graph is called covered if it has been in all of the three states: active, stalled and flushed. An edge is *active* when it is used to transfer an operation in a clock cycle. An edge is *stalled* if it does not transfer an operation in a clock cycle from parent node to children node. An edge is *flushed* if the parent node is flushed due to the exception in the children node. The edge coverage conditions are redundant if a node has only one children. However, if a node has multiple children (or parent), edge coverage conditions are necessary.

Our test generation algorithm traverses the pipeline graph and generates properties based on the graph coverage described above. Consider the test generation for a feedback path (edge) from *MUL7* to *IALU* in Figure 3. To generate a test for making the feedback path *active*, two properties are generated: i) make the node *MUL7* active in clock cycle *t*, and ii) make the node *IALU* active in clock cycle *(t+1)*. This would lead to a test program that has a multiply operation followed by six NOPs (no operation), and finally an add operation.

4 A Case Study

In a case study we successfully applied the proposed methodology to the DLX processor [6]. We have chosen DLX processor since it has been well studied in academia and contains many interesting features such as, fragmented pipelines and multicycle units that are representative of many commercial pipelined processor architectures such as TI C6x, PowerPC and MIPS R10K. Figure 3 shows the graph model of the DLX processor. The DLX architecture has five pipeline stages: fetch (IF), decode (ID), execute, memory (MEM), and writeback (WB). First, we present the test program generation results for the DLX processor. Next, we describe a test generation scenario using an illustrative example to demonstrate the efficiency of our technique.

4.1 Test Generation Results

This section describes the number of test cases generated for the DLX processor using the functional coverage described in Section 3.4. The DLX processor shown in Figure 3 has 20 nodes and 24 edges (except feedback paths). We have described 91 instructions of the DLX processor [6].

Table 1. Number of Test Programs in Different Categories

Node Coverage				Edge Coverage		
Active	Stalled	Flushed	Exception	Active	Stalled	Flushed
91	20	20	20	24	24	24

Table 1 shows the number of test programs generated for node and edge coverage of the DLX processor. Although, 20 testcases would suffice for the *active* node coverage, we cover all the instructions. Also, there are many ways of making a node stalled, flushed or in exception condition. We chose one such scenario. If we consider all possible scenarios, the number of test programs will increase. In this case, our algorithm generated 223 test programs in 91 seconds on a 333 MHz Sun UltraSPARC-II with 128M RAM.

Table 2. Reduced Number of Test Programs

Node Coverage				Edge Coverage		
Active	Stalled	Flushed	Exception	Active	Stalled	Flushed
4	14	2	20	4†	14† + 3	2†

As mentioned earlier, some of the test programs are redundant. For example, since there are four pipeline paths, we need only four test programs that exercises the four paths. These four test programs will make all the nodes *active*. Similarly, if we assume VLIW DLX, the decode node will be stalled if any one of its four children is stalled. Furthermore, if MEM node is stalled, all of its four parents will also be stalled. This implies that we need only 14 testcases for node stalling. Likewise, if the MEM node is in exception, the instructions in all the previous nodes will be flushed. Hence, we need only 2 testcases for flushing. Finally, some of the node coverage testcases also satisfies the edge coverage. We need a total of 43 test programs in this case. Table 2 shows the number of reduced test programs in different categories.

4.2 Test Program Generation: An Example

In this section we describe our test generation approach using the following example. We use this example to compare the performance of our test generation algorithm with previously published results [12].

Example 1: *Consider a fragment of the DLX pipeline containing three internal registers of the division unit (DIV) as shown in Figure 4. 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.,

† Same testcases as in the node coverage.

$A_{in} = divInst.src1$ 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., $result = C_{out}$.

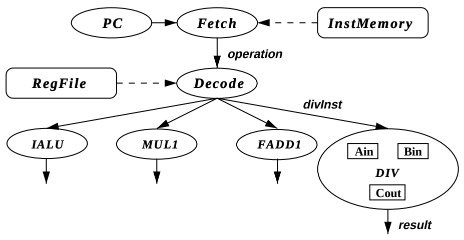

Figure 4. A fragment of the DLX architecture

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 [8]. 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)));
```

Mishra et al. [12] applied this property on the complete description of the DLX processor to generate the required test program. They used a 359 MHz Sun UltraSPARC-II with 2048M RAM and the test generation time was 75.4 seconds. We do not have access to such a machine. We applied this property using a 333 MHz Sun UltraSPARC-II with 128M RAM and it took 375.98 seconds to generate the test program. One of the reason for this difference is the lower RAM size of our machine. The number of allocated BDD nodes is 1928568.

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.

As mentioned earlier, if the property is applied to the complete description of the processor, SMV takes 375.98 seconds and 1928568 BDD nodes to generate the counterexample. Clearly, our technique reduced the test generation time and the required BDD size by an order of magnitude.

5 Conclusions

Functional validation is widely acknowledged as a major bottleneck in microprocessor design. Specification driven validation is a promising approach. In this paper, we present a graph coverage based functional test program generation technique for pipelined processors. Our methodology accepts architecture description language (ADL) specification of the processor as input. A graph model of the pipelined processor is generated from the ADL specification. We have defined the functional coverage of the pipeline behavior in terms of the graph coverage. We have presented a test program generation algorithm that traverses the pipeline graph to generate test programs based on the coverage metric. The algorithm breaks one processor level property into multiple module level properties and applies them. Our technique reduced the test generation time and the required BDD size by an order of magnitude.

Currently, we apply these tests on the cycle-accurate structural simulator of the architecture. Our future work includes application of these test programs on the RTL description for functional validation of pipelined processors.

References

[1] A. Aharon et al. Test Program Generation for Functional Verification of PowerPC Processors in IBM. *DAC*, 1995.

[2] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. *CAV*, 1994.

[3] D. Campenhout et al. High-Level Test Generation for Design Verification of Pipelined Microprocessors. *DAC*, 1999.

[4] L. Chen et al. A Scalable Software-Based Self-Test Methodology for Programmable Processors. *DAC*, 2003.

[5] A. Halambi et al. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. *DATE*, 1999.

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

[7] P. Ho et al. Formal verification of pipeline control using controlled token nets and abstract interpretation. *ICCAD*, 1998.

[8] www.cs.cmu.edu/~modelcheck. *Symbolic Model Verifier*.

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

[10] R. Jhala and K. L. McMillan. Microarchitecture Verification by Compositional Model Checking. *CAV*, 2001.

[11] W. Lai and K. Cheng. Instruction-Level DFT for Testing Processor and IP Cores in System-on-a-Chip. *DAC*, 2001.

[12] P. Mishra and N. Dutt. Automatic Functional Test Program Generation for Pipelined Processors using Model Checking. *HLDVT*, 2002.

[13] P. Mishra et al. Functional Abstraction driven Design Space Exploration of Heterogeneous Programmable Architectures. *ISSS*, 2001.

[14] J. Sawada and J. W.A. Hunt. Trace Table based Approach for Pipelined Microprocessor Verification. *CAV*, 1997.

[15] J. Shen et al. Functional Verification of the Equator MAP1000 Microprocessor. *DAC*, 1999.

[16] J. Skakkebaek et al. Formal Verification of Out-of-order Execution using Incremental Flushing. *CAV*, 1998.

[17] S. Thatte and J. Abraham. Test Generation for Microprocessors. *IEEE Transactions on Computers*, C-29(6), 1980.

[18] S. Ur and Y. Yadin. Micro architecture coverage directed generation of test programs. *DAC*, 1999.