UNIVERSITY OF CALIFORNIA,
IRVINE

Specification-driven Validation of Programmable Embedded Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Prabhat Kumar Mishra

Dissertation Committee:
Professor Nikil Dutt, Chair
Professor Rajesh Gupta
Professor Alex Nicolau

2004

The dissertation of Prabhat Kumar Mishra
is approved and is acceptable in quality
and form for publication on microfilm:

_____

_____

_____

Committee Chair

University of California, Irvine
2004

To my parents and my wife.

Contents

List of Figures vii

List of Tables ix

Acknowledgments x

Curriculum Vitae xi

Abstract of the Dissertation xii

1 Introduction 1
- 1.1 Traditional Bottom-Up Validation Flow . . . 2
- 1.2 Proposed Top-Down Validation Methodology . . . 4
- 1.3 Thesis Contributions . . . 5
- 1.4 Thesis Organization . . . 8

2 Architecture Specification 9
- 2.1 Architecture Description Languages . . . 9
  - 2.1.1 Behavioral ADLs . . . 11
  - 2.1.2 Structural ADLs . . . 11
  - 2.1.3 Mixed ADLs . . . 12
- 2.2 Specification using EXPRESSION ADL . . . 13
  - 2.2.1 Processor Specification . . . 15
  - 2.2.2 Coprocessor Specification . . . 16
  - 2.2.3 Memory Subsystem Specification . . . 18
  - 2.2.4 Specification of Interrupts and Exceptions . . . 19
- 2.3 Chapter Summary . . . 21

3 Validation of Specification 23
- 3.1 Validation of Static Behavior . . . 24
  - 3.1.1 Graph-based Modeling of Pipelines . . . 25
  - 3.1.2 Validation of Pipeline Specifications . . . 29

3.1.3 Experiments . . . 42
3.2 Validation of Dynamic Behavior . . . 46
3.2.1 FSM-based Modeling of Processor Pipelines . . . 47
3.2.2 Validation of Dynamic Properties . . . 52
3.2.3 A Case Study . . . 58
3.3 Chapter Summary . . . 60

4 Model Generation using Functional Abstraction 62
4.1 Survey of Contemporary Architectures . . . 63
4.1.1 Summary of Architectures Studied . . . 63
4.1.2 Similarities and Differences . . . 64
4.2 Functional Abstraction . . . 67
4.2.1 Structure of a Generic Processor . . . 67
4.2.2 Behavior of a Generic Processor . . . 72
4.2.3 Structure of a Generic Memory Subsystem . . . 72
4.2.4 Generic Controller . . . 73
4.2.5 Interrupts and Exceptions . . . 74
4.3 Reference Model Generation . . . 75
4.4 Design Space Exploration . . . 79
4.4.1 Simulator Generation and Exploration . . . 79
4.4.2 Hardware Generation and Exploration . . . 90
4.5 Chapter Summary . . . 94

5 Specification-driven Validation 96
5.1 Design Validation . . . 98
5.1.1 Property Checking using Symbolic Simulation . . . 99
5.1.2 Equivalence Checking . . . 101
5.2 Experiments . . . 103
5.2.1 Property Checking of a Memory Management Unit . . . 103
5.2.2 Equivalence Checking of the DLX Architecture . . . 106
5.3 Chapter Summary . . . 107

6 Functional Test Generation 109
6.1 Test Generation using Model Checking . . . 109
6.1.1 Test Generation Methodology . . . 110
6.1.2 A Case Study . . . 113
6.2 Functional Coverage driven Test Generation . . . 117
6.2.1 Functional Fault Models . . . 118
6.2.2 Functional Coverage Estimation . . . 121
6.2.3 Test Generation Techniques . . . 121
6.2.4 A Case Study . . . 126
6.3 Chapter Summary . . . 130

7 Conclusions and Future Work **132**
7.1 Conclusions 132
7.2 Future Research Directions 134

Bibliography **137**

List of Figures

1.1 An example embedded system 2
1.2 Bottom-up validation flow for programmable embedded systems . . . 3
1.3 Proposed specification-driven validation methodology 5

2.1 ADL-driven design space exploration 10
2.2 Block level description of an example architecture 13
2.3 Pipeline level description of the DLX processor shown in Figure 2.2 . 14
2.4 Processor specification using EXPRESSION ADL 16
2.5 Coprocessor specification using EXPRESSION ADL 17
2.6 Memory subsystem specification using EXPRESSION ADL 18
2.7 Specification of division_by_zero exception 19
2.8 Specification of illegal_slot_instruction exception 20
2.9 Specification of machine_reset exception 20
2.10 Specification of interrupts 21

3.1 Validation of pipeline specifications 25
3.2 An example architecture 26
3.3 A fragment of the behavior graph 28
3.4 An example processor with false pipeline paths 32
3.5 An example processor with false data-transfer paths 35
3.6 The DLX architecture 43
3.7 ADL driven validation of pipeline specifications 46
3.8 A fragment of the processor pipeline 48

3.9 Automatic validation frameworks 57

4.1 A fetch unit example 68
4.2 Modeling of RenameRegister function using sub-functions 69
4.3 Modeling of MAC operation 72
4.4 Modeling of associative cache function using sub-functions 73
4.5 Examples of distributed and centralized controllers 74
4.6 Mapping between *MACcc* and generic instructions 77
4.7 Simulation model generation for the DLX architecture 78
4.8 Architecture exploration framework 80
4.9 Cycle counts for different graduation styles 81
4.10 Functional unit vs. coprocessor 82
4.11 Memory exploration results for GSR 87
4.12 Energy performance tradeoff for Compress 88
4.13 Energy performance tradeoff for MatMult 89
4.14 Energy performance tradeoff for Laplace 90
4.15 The application program 91
4.16 Pipeline path exploration 92
4.17 Pipeline stage exploration 93
4.18 Instruction-set exploration 94

5.1 Top-down validation methodology 97
5.2 Test vectors for validation of an *AND* gate 99
5.3 Compare point matching between reference and implementation design 102
5.4 TLB block diagram 105

6.1 Test program generation methodology 111
6.2 A fragment of the DLX architecture 115

List of Tables

3.1 Specification validation time for different architectures 42
3.2 Summary of property violations during DSE 45
3.3 Validation of in-order execution by two frameworks 60

4.1 Processor-memory features of different architectures. *R4K: MIPS R4000, SA: StrongArm, 56K: Motorola 56K, c5x: TI C5x, c6x: TI C6x, MA: MAP1000A, SC: Starcore, R10: MIPS R10000, MP: Motorola MPC7450, U3: SUN UltraSparc IIi, α64: Alpha 21364, IA64: Intel IA-64* 65
4.2 A list of common sub-functions 70
4.3 Benchmarks 84
4.4 The memory subsystem configurations 85

5.1 Validation of the DLX implementation using equivalence checking . . 107

6.1 Number of test programs in different categories 114
6.2 Reduced number of test programs 114
6.3 Test programs for validation of DLX architecture 129
6.4 Test programs for validation of LEON2 processor 130

Acknowledgments

First of all, I would like to thank my advisor Prof. Nikil Dutt for his guidance and support throughout my graduate studies. Without the numerous discussions and brainstorms with him, the results presented in this thesis would never have existed. I am grateful to Prof. Alex Nicolau for his guidance and encouragement during last four years. I would also like to thank Prof. Rajesh Gupta for his valuable comments and suggestions on my research.

This thesis is the result of many collaborations. I would like to acknowledge the contributions of Jonas Astrom, Dr. Peter Grun, Ashok Halambi, Arun Kejariwal, Mahesh Mamidipaka, Dr. Frederic Rousseau, Prof. Sandeep Shukla, and Prof. Hiroyuki Tomiyama. I would also like to thank Dr. Magdy Abadir and Dr. Narayanan Krishnamurthy for their help in my research work.

I am thankful to many people in the Center for Embedded Computer Systems (CECS) for making my journey a memorable one. In particular, I would like to thank Melanie Sanders for her help and understanding throughout my graduate life. I would also like to thank all my colleagues in CECS, including Ana Azevedo, Sudarshan Banerjee, Nikhil Bansal, Partha Biswas, Radu Cornea, Paolo DAlberto, Sumit Gupta, Ilya Issenin, Dan Nicolaescu, Sudeep Pasricha, Mehrdad Reshadi, Nick Savoiu, Srikanth Srinivasan, and Aviral Srivastava.

Last but not least, I am grateful to my parents and my wife for their love, encouragement, and understanding. It would be impossible for me to express my gratitude towards them in mere words. I dedicate this thesis to them.

Curriculum Vitae

Prabhat Mishra

Education

2004 **Ph.D.** in Computer Science, University of California, Irvine, USA
1996 **M.Tech.** in Computer Science, Indian Institute of Technology, Kharagpur
1994 **B.E.** in Computer Science, Jadavpur University, Calcutta, India

Research and Work Experience

1999–2004	Graduate Researcher	University of California, Irvine, USA
2001, 2002	Summer Research Intern	Somerset Design Center, Motorola, USA
2000	Summer Research Intern	IA-64 Performance Group, Intel, USA
1998–1999	Senior R&D Engineer	Synopsys, Bangalore, India
1997–1998	Senior Software Engineer	Sasken, Bangalore, India
1996–1997	Software Design Engineer	Texas Instruments, Bangalore, India

Selected Publications

1. P. Mishra and N. Dutt, *"Modeling and Validation of Pipeline Specifications"*, ACM Transactions on Embedded Computing Systems (TECS), 3(1), 114–139, 2004.
2. P. Mishra, M. Mamidipaka, and N. Dutt, *"Processor-Memory Co-Exploration using an Architecture Description Language"*, ACM TECS, 3(1), 140–162, 2004.
3. P. Mishra, N. Dutt, and H. Tomiyama, "Towards Automatic Validation of Dynamic Behavior in Pipelined Processor Specifications", Kluwer Design Automation for Embedded Systems (DAES), 8(2/3), pages 249-265, 2003.
4. P. Mishra, N. Dutt, N. Krishnamurthy, and M. Abadir, *"A Top-Down Methodology for Validation of Microprocessors"*, IEEE Design & Test of Computers, 2004.
5. P. Mishra and N. Dutt, *"Graph-based Functional Test Program Generation for Pipelined Processors"*, Design Automation and Test in Europe (DATE), 2004.
6. P. Mishra, A. Kejariwal, and N. Dutt, *"Synthesis-driven Exploration of Pipelined Embedded Processors"*, International Conference on VLSI Design, 921–926, 2004.
7. M. Reshadi, P. Mishra, and N. Dutt, *"Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation"*, Design Automation Conference (DAC), 758–763, 2003.
8. M. Reshadi, N. Bansal, P. Mishra, and N. Dutt, *"An Efficient Retargetable Framework for Instruction-Set Simulation"*, Intl. Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 13-18, 2003. **Best Paper Award**

Abstract of the Dissertation

Specification-driven Validation of Programmable Embedded Systems

by

Prabhat Kumar Mishra

Doctor of Philosophy in Information and Computer Science
University of California, Irvine, 2004
Professor Nikil Dutt, Chair

Validation of programmable embedded systems, consisting of processor cores, co-processors, and memory subsystems, is one of the major bottleneck in current System-on-Chip (SOC) design methodology. One of the most important problems in validation of such systems is the lack of a golden reference model. As a result, many existing validation techniques employ a bottom-up approach to design verification, where the functionality of an existing architecture is, in essence, reverse-engineered from its implementation. This thesis presents a top-down validation methodology that complements the existing bottom-up approaches. It leverages the system architect's knowledge about the behavior of the design through architecture specification. We have developed validation techniques to ensure that the static and dynamic behaviors of the specified architecture is well formed. The validated specification is used as a golden reference model. A major challenge in top-down validation methodology is the ability to generate executable models from the specification for a wide variety of programmable architectures. We have developed a functional abstraction technique that enables specification-driven model generation for simulation, hardware generation, and property checking. The generated simulator and hardware models are used for design space exploration of programmable architectures. We have explored two top-down validation scenarios: design validation and test generation. First, the generated hardware is used as a reference model to verify the hand-written implementation using a combination of symbolic simulation and equivalence checking. Second, we have proposed a functional coverage based test generation technique for validation of pipelined processor architectures. The experiments demonstrate the utility of the specification-driven validation methodology for programmable embedded systems.

Chapter 1

Introduction

Embedded systems run the computing devices hidden inside a vast array of everyday products and appliances such as cell phones, toys, handheld PDAs, cameras, and microwave ovens. Cars are full of them, as are airplanes, satellites, and advanced military and medical equipments. As applications grow increasingly complex, so do the complexities of the embedded computing devices. Figure 1.1 shows an example embedded system, consisting of programmable components including processor cores, coprocessors and a memory subsystem. The memory subsystem contains memory components such as cache hierarchies and scratch-pad SRAM. Depending on the application domain, the embedded system can have application specific hardwares, interfaces, controllers, and peripherals. In this thesis, we refer to the programmable components, consisting of the processor core, coprocessors, and memory subsystem, as *programmable embedded systems*. We also refer to them as programmable architectures.

Embedded systems are typically designed for dedicated tasks or application domains. The design of such embedded systems often begins with the specification of the application, as well as the definition of the architecture onto which the application is mapped. Validation of programmable embedded systems is one of the most important and complex tasks in embedded systems design. A significant bottleneck in the validation of such systems is the lack of a golden reference model. As a result, many existing approaches employ a bottom-up validation approach by using a combi-

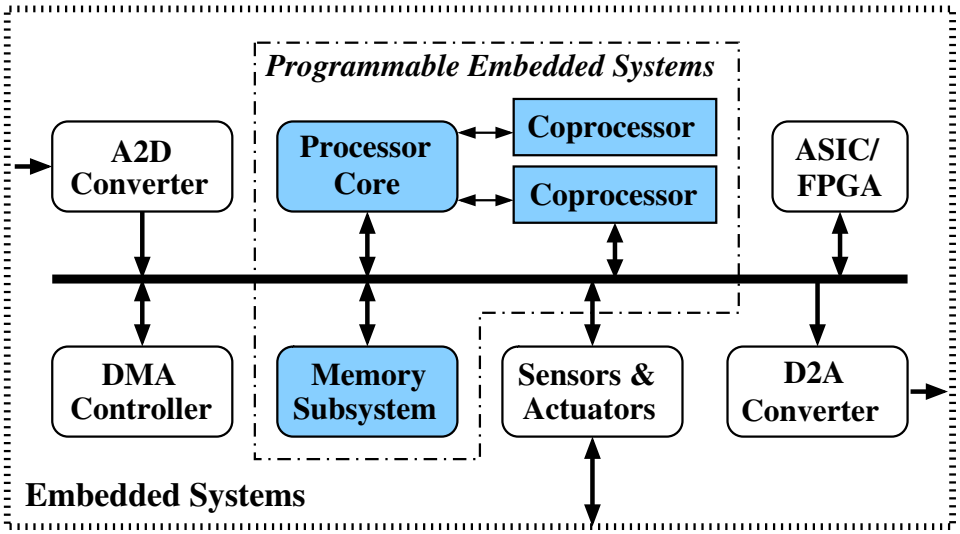


Figure 1.1: An example embedded system

nation of simulation techniques and formal methods. This thesis presents a top-down validation methodology for programmable embedded systems that complements the existing bottom-up techniques. We consider validation of systems with single processor core, coprocessors and memories. Today's embedded systems employ deeply pipelined processor architectures to meet desired system performance. One of the major challenges in validation of such systems is the verification of processor pipelines. The main focus of this thesis is the validation of pipelined processor architectures.

This chapter provides an overview of the problems that will be addressed in the rest of the thesis and outlines a brief summary of the thesis contributions.

1.1 Traditional Bottom-Up Validation Flow

Figure 1.2 shows a traditional architecture validation flow. In the current validation methodology, the architect prepares an informal specification of the programmable embedded systems in the form of an English document. The logic designer implements the modules at the register-transfer level (RTL). The *RTL design* is validated using a combination of simulation techniques and formal methods. Simulation is the most widely used form of microprocessor validation using random (or

pseudo-random) testcases [1, 12, 35, 77, 91]. Model checking is applied on the high-level description of the design abstracted from the RTL implementation [23, 37]. Formal verification is performed by describing the system using a formal language [11, 15, 24, 75, 80, 81, 92]. The specification for the formal verification is derived from the architecture description. The implementation for the formal verification can be derived either from the architecture specification or from the abstracted design. The validated *RTL design* is used as a golden reference model for future design modifications. Several design transformations (including synthesis) are applied on the *RTL design*. The modified design is validated against the *RTL design* using equivalence checking.

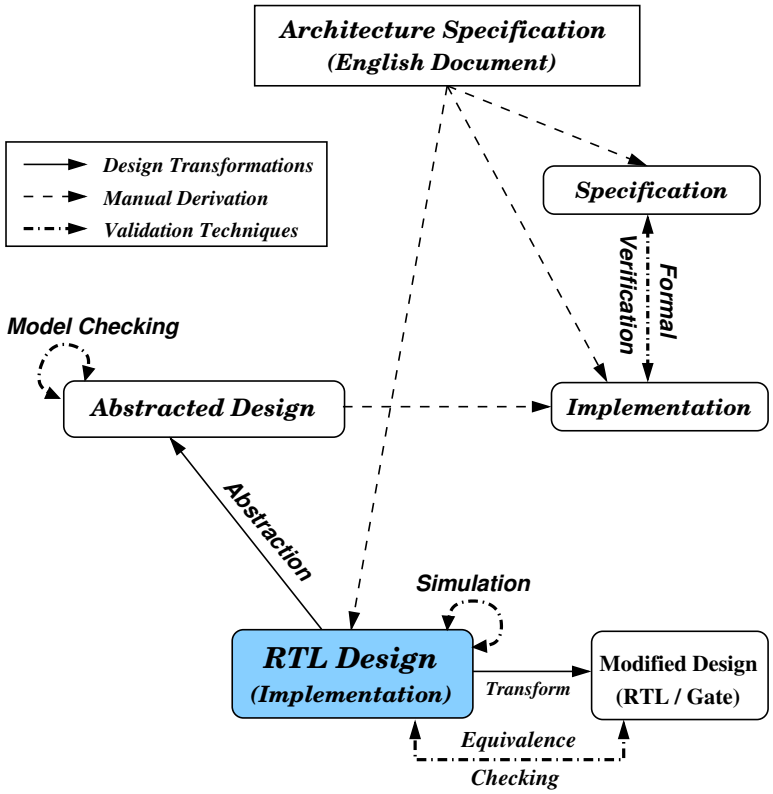


Figure 1.2: Bottom-up validation flow for programmable embedded systems

The existing processor validation techniques can be divided into two categories depending on the models used for specification and implementation. The techniques in the first category do not verify the actual design implemented by the logic designers.

Instead, they verify the implementation written in a formal language that is either abstracted from the actual design or written manually. Hence, this verification does not uncover the bugs in the actual design. The traditional formal verification techniques are in this category. The techniques in the second category are applied on the actual RTL implementation. A significant bottleneck in these techniques is the lack of a golden reference model. As a result, they derive the specification model from the actual implementation by reverse-engineering methods. The simulation based techniques are in this category. The existing techniques employ a bottom-up approach to validation, where the functionality of an existing processor is, in essence, reverse-engineered from its RTL implementation.

1.2 Proposed Top-Down Validation Methodology

The validation technique proposed in this thesis is complementary to the existing bottom-up approaches. Our approach leverages the system architect's knowledge about the behavior of the pipelined architecture, through Architecture Description Language (ADL) constructs, and thus allows a top-down approach to architecture validation.

Figure 1.3 shows our top-down validation methodology. The first step is to capture the programmable architecture using the EXPRESSION ADL [20]. The static and dynamic behaviors of the architecture specification are validated to ensure that the specification is well formed. The validated ADL specification is used to generate various executable models including simulator, hardware, and validation models. This thesis explores three top-down validation scenarios using the generated models: *design space exploration*, *design validation*, and *test generation*.

The generated hardware model (by **HDLGen**) and simulator (by **SimGen**) are used for design space exploration of programmable architectures for the given set of application programs under various design constraints such as area, power, and performance. The generated hardware is also used as a reference model for verifying the hand-written implementation (RTL Design) using a combination of symbolic simulation and equivalence checking. Finally, the specification is used to generate

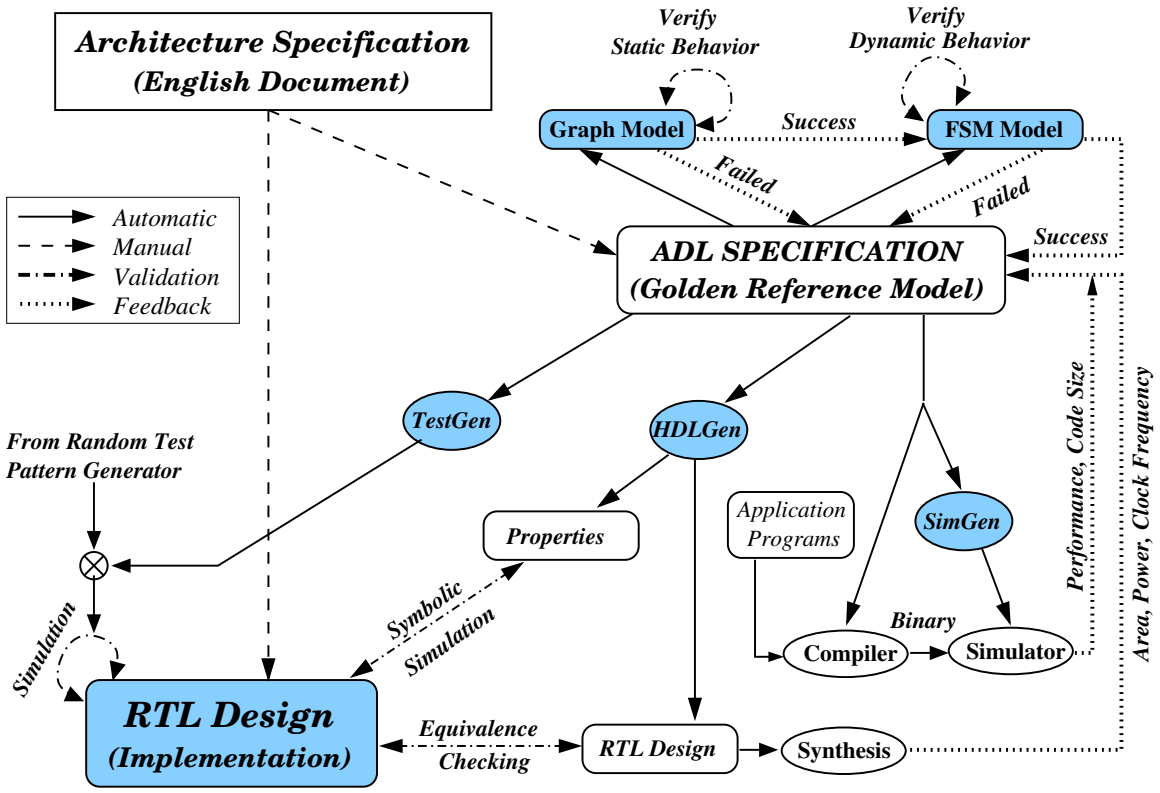


Figure 1.3: Proposed specification-driven validation methodology

functional test programs (by **TestGen**) based on the functional coverage of pipelined architectures. The generated test programs are used to simulate the implementation, and complement the tests generated by the existing techniques such as a random test pattern generator.

1.3 Thesis Contributions

This thesis makes two major contributions: *validation of the architectural specification*, and *specification driven validation* of programmable embedded systems. We explore three specification-driven validation scenarios: *model generation and exploration*, *top-down design validation*, and *functional test program generation*. Specification validation, model generation, and exploration studies are conducted on programmable architectures with a pipelined processor, coprocessors, and a memory

subsystem. However, design validation and test generation techniques are applied mainly on pipelined processor architectures.

We have developed validation techniques to ensure that the architectural specification is well formed by analyzing the static behavior of the architecture [52, 56, 70]. A novel feature of our approach is the ability to model the pipeline structure and behavior of the processor, coprocessor, as well as the memory subsystem using a graph-based model. Based on this model we present algorithms to ensure that the specification is well formed by verifying several architectural properties, such as connectedness, false pipeline paths, completeness, and finiteness. We have also developed techniques to verify the execution style of the processor architecture specified in the ADL. The dynamic behavior is verified by analyzing the instruction flow in the pipeline using a finite-state machine (FSM) based model to validate several important architectural properties such as determinism and in-order execution in the presence of hazards and multiple exceptions [51, 60, 69]. The validated ADL specification is used as a golden reference model for specification-driven validation of programmable embedded systems.

A major challenge in a top-down validation methodology is the ability to generate executable models from the specification for a wide variety of programmable architectures including RISC (Reduced Instruction Set Computing), DSP (Digital Signal Processing), VLIW (Very Long Instruction Word), and superscalar architectures. We developed a functional abstraction approach by studying the similarities and differences of each architectural feature in various architecture domains. Based on our observations we have defined necessary generic functions, sub-functions, and computational environment needed to capture a wide variety of programmable architectures. Our functional abstraction technique enables generation of models for simulation, hardware generation, and property checking from the ADL specification. We have developed a technique for retargetable simulator generation using functional abstraction [57]. The generated simulators are used for functional validation and design space exploration of programmable architectures [61, 65, 66]. Simulation-based exploration may answer questions concerning the instruction set, the performance of an algorithm and the required size of memory and registers. However, it is necessary

to generate hardware model to determine the required silicon area, clock frequency, and power consumption of the specified architectures. We have also developed a technique for specification-driven hardware generation [62, 63]. Our design space exploration results demonstrate the power of reuse in composing architectures using functional abstraction primitives allowing for a reduction in the time for specification, model generation, and exploration. by an order of magnitude.

The generated hardware is also used as a reference model for verifying the hand-written RTL implementation [55]. An important aspect of our methodology is the ability to perform both model (property) checking and equivalence checking depending on the generated reference model. To verify that the implementation satisfies certain properties, our framework generates the intended properties. We use a symbolic simulator to perform property checking. Our framework generates the RTL description of the pipelined processor to enable equivalence checking with the hand-written implementation.

We have developed two specification-driven test generation techniques: model checking based and functional coverage based. The first technique uses a model checker to generate test programs [50, 54]. The processor model is generated from the ADL specification. Properties are generated based on the coverage of the processor pipeline. The generated properties are applied on the processor model using the SMV model checker [28]. The generated counterexamples are converted into test programs consisting of instruction sequences. We have also developed a functional coverage based test generation technique for pipelined architectures [53]. A general graph-theoretic model is developed that can capture the structure and behavior (instruction-set) for a wide variety of pipelined processors. We have proposed a functional fault model that is used to define the functional coverage of pipelined processors. We have also developed test generation procedures that accept the graph model of the architecture as input and generate test programs to detect all the faults in the functional fault model. Our experimental results demonstrate that the number of test programs generated by our approach to obtain a fault coverage is an order of magnitude less than those generated by random or constrained-random test generation techniques.

1.4 Thesis Organization

The organization of the thesis is as follows. Chapter 2 surveys the existing ADLs that can be used to specify the programmable architectures. This chapter also presents the EXPRESSION ADL that is used in our validation framework to specify processor, coprocessor, and memory architectures.

Chapter 3 presents the techniques for validating the architecture specification. These techniques verify the static and dynamic behaviors of the specified architecture. The validated specification is used as a reference model in three top-down validation scenarios: model generation and exploration, design validation, and test generation.

The functional abstraction technique is presented in Chapter 4. It enables automatic generation of models for simulation, hardware generation, and validation for a wide variety of programmable architectures. This chapter also presents exploration experiments using the generated simulator and hardware models.

In Chapter 5, the generated hardware is used as a reference model for verifying the hand-written RTL implementation using a combination of symbolic simulation and equivalence checking.

Chapter 6 presents two specification-driven test generation techniques. The first technique enables functional test program generation using model checking. The second technique generates test programs based on functional coverage of the pipelined processor architectures.

Finally, Chapter 7 contains a summary of the thesis and a discussion of future research directions.

Chapter 2

Architecture Specification

The first step in a top-down validation methodology is to capture the programmable architecture using a specification language. The language should be powerful enough to specify the wide spectrum of contemporary processor, coprocessor, and memory features. On the other hand, the language should be simple enough to allow correlation of the information between the specification and the architecture manual.

This chapter is organized as follows. Section 2.1 introduces the notion of an Architecture Description Language (ADL) and surveys the existing ADLs in terms of their specification capabilities. Section 2.2 describes architecture specification using EXPRESSION ADL [20]. Finally, Section 2.3 summarizes the chapter.

2.1 Architecture Description Languages

The phrase *Architecture Description Language* (ADL) is used in the context of designing both software and hardware architectures. Software ADLs are used for representing and analyzing software architectures ([14], [47]). It captures the behavioral specifications of the components and their interactions that comprise the software architecture. On the other hand, hardware ADLs (such as processor ADLs) capture the structure (hardware components and their connectivity), and the behavior (instruction-set) of processor architectures. In this thesis the term ADL will refer to hardware architecture description language.

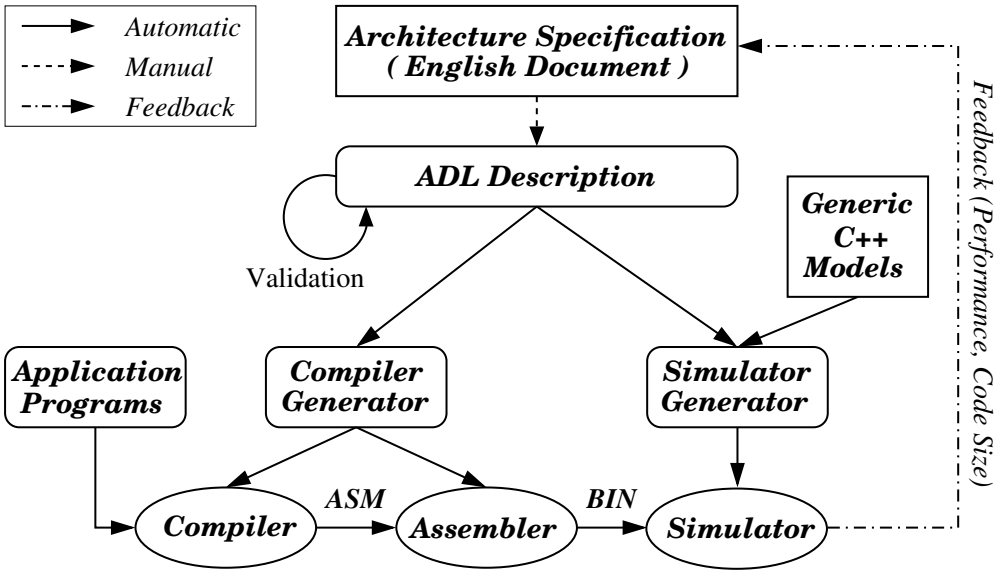


Figure 2.1: ADL-driven design space exploration

Traditionally, ADLs are used for early exploration of programmable embedded systems. Embedded systems present a tremendous opportunity to customize designs by exploiting the application behavior. ADLs enable exploration of programmable architectures for a given set of application programs under the design constraints such as area, power and performance.

Figure 2.1 shows a traditional ADL-driven exploration flow. The ADL is used to specify the processor, coprocessor and memory architectures. The software toolkit including the compiler, simulator and assembler is generated from the ADL specification, and the feedback is used to modify the architecture. Although, ADL-driven exploration is extensively used in both academia (nML [17], ISDL [19], EXPRESSION [20], Valen-C [34], MIMOLA [45], Sim-nML [74], and LISA [96]), and industry (ARC [5], Axys [6], RADL [79], Target [87], Tensilica [88], LISATek [27], and MDES [90]), to the best of our knowledge, there has not been any effort in validating the ADL specification. It is necessary to validate the ADL specification of the architecture to ensure the correctness of both the architecture specified, as well as the generated software toolkit. Chapter 3 presents specification validation techniques for pipelined processors.

Traditionally, ADLs have been classified into two categories depending on whether they primarily capture the behavior (instruction set) or the structure of the processor. Recently, many ADLs have been proposed that capture both the structure and the behavior of the architecture.

2.1.1 Behavioral ADLs

nML [17] and ISDL [19] are examples of behavior-centric ADLs. In nML, the processor's instruction-set is described as an attributed grammar with the derivations reflecting the set of legal instructions. nML has been used by the retargetable code generation environment CHESS [42] to describe DSP and ASIP (Application Specific Instruction set Processor) architectures. In ISDL, constraints on parallelism are explicitly specified through illegal operation groupings. This could be tedious for complex architectures like DSPs which permit operation parallelism (e.g. Motorola 56K) and VLIW machines with distributed register files (e.g. TI C6X). The retargetable compiler system by Yasuura et al. [34] produces code for RISC architectures starting from an instruction set processor description, and an application described in Valen-C.

Many behavioral ADLs share one common feature: a hierarchical instruction set description based on attribute grammars [36]. This feature greatly simplifies the instruction set description by exploiting the common components between operations. However, the lack of detailed pipeline and timing information prevents the use of these languages as an extensible architecture model. Information required by resource-based scheduling algorithms cannot be obtained directly from the description. Also, it is impossible to generate cycle accurate simulators based on the behavioral descriptions without some assumptions on the architecture's control behavior, i.e., an implied architecture template has to be used.

2.1.2 Structural ADLs

MIMOLA [45] and UDL/I [2] are examples of ADLs that primarily capture the structure of the processor wherein the net-list of the target processor is described

in a HDL (Hardware Description Language) like language. One advantage of this approach is that the same description is used for both processor synthesis and code generation. The target processor has a micro-coded architecture. Using MIMOLA, the net-list description is used to extract the instruction set [44, 45], and produce the code generator. UDL/I [2] is used for describing processors at an RT-level on a per-cycle basis. The instruction-set is automatically extracted from the UDL/I description [3], and is then used for generation of a compiler and a simulator.

In general, structural ADLs enable flexible and precise micro-architecture descriptions. The same description can be used for hardware synthesis, test generation, simulation and compilation. However, it is difficult to extract the instruction-set for retargetable compilation. The instruction set information is buried under enormous micro-architectural detail. These ADLs are more suited for hardware design than for retargetable software-toolkit generation.

2.1.3 Mixed ADLs

More recently, languages that capture both the structure and the behavior of the processor, as well as detailed pipeline information have been proposed (EXPRESSION [20], RADL [79], FLEXWARE [73], MDes [90], and LISA [96]). The main characteristic of LISA is the operation-level description of the pipeline. RADL [79] is an extension of the LISA approach that focuses on explicit support of detailed pipeline behavior to enable generation of production quality cycle-accurate and phase-accurate simulators. FLEXWARE [73] and MDes [90] have a mixed-level structural/behavioral representation. FLEXWARE contains the CODESYN code-generator and the Insulin simulator for ASIPs. The simulator uses a VHDL model of a generic parameterizable machine. The application is translated from the user-defined target instruction set to the instruction set of this generic machine. The MDes [90] language used in the Trimaran system is a mixed-level ADL, intended for exploration of parameterized VLIW architectures. Information is broken down into sections (such as format, resource-usage, latency, operation, and register), based on a high-level classification of the information being represented. However, MDes allows only a restricted retargetability of

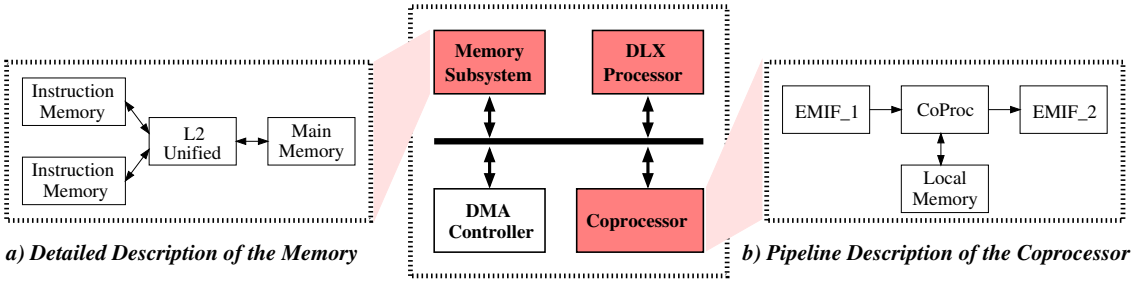


Figure 2.2: Block level description of an example architecture

the simulator to the HPL-PD processor family. MDes permits the description of the memory system, but is limited to the traditional memory architectures consisting of register files and caches. It will be difficult to describe novel memory subsystems that includes efficient memory modules (such as partitioned register files, scratch-pad SRAM, stream buffer, SDRAM, DDRAM, and RAMBUS) exhibiting a heterogeneous set of features (such as page-mode, burst-mode and pipelined accesses).

The EXPRESSION ADL also follows a mixed-level approach to facilitate DSE. Furthermore, it provides support for specification of novel memory subsystems. It avoids explicit representation of the reservation tables[1] by extracting them from the structural description [18]. The ADL is used to drive the generation of both compiler [21] and simulator [40].

2.2 Specification using EXPRESSION ADL

Our validation framework uses the EXPRESSION ADL [20] to specify processor, coprocessor, and memory architectures. The EXPRESSION ADL follows a mixed-level approach to facilitate specification of a wide range of programmable embedded systems. We illustrate the use of the EXPRESSION ADL to describe a simple multi-issue architecture consisting of a processor, a co-processor and a memory subsystem.

Figure 2.2 shows the block level description of a simple architecture. This level of detail is available in a typical architecture manual. Typically, pipeline level de-

[1]Reservation Tables (RTs) have been used to detect conflicts between instructions that simultaneously access the same architectural resource.

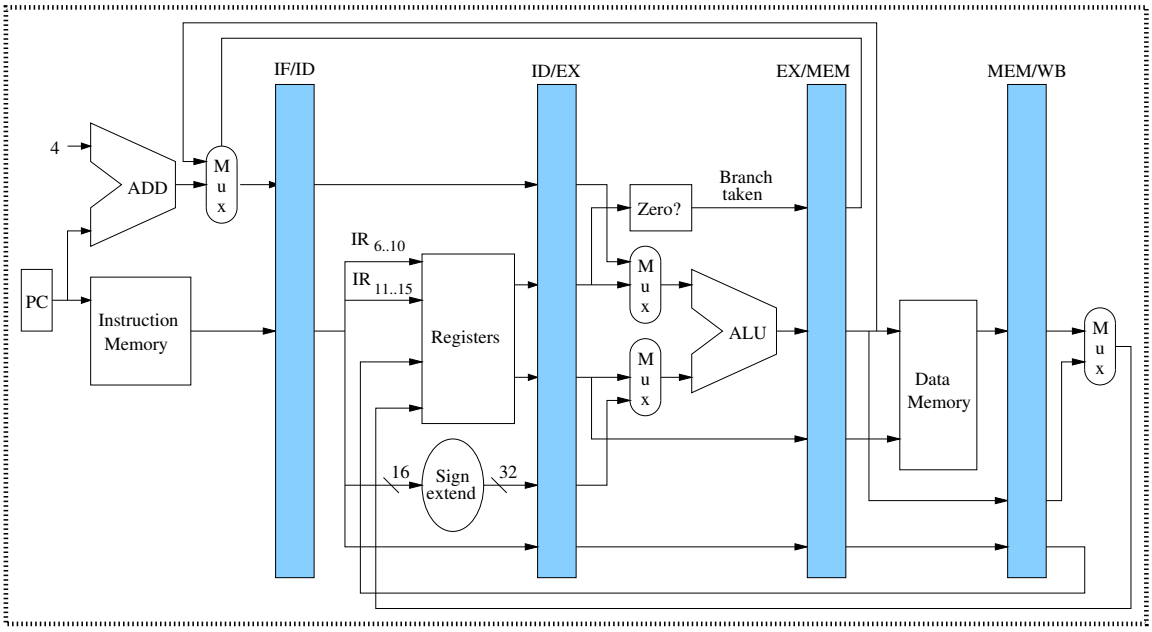


Figure 2.3: Pipeline level description of the DLX processor shown in Figure 2.2

tails are available in a micro-architecture manual. For example, Figure 2.2(a) and Figure 2.2(b) show the detailed description of the memory subsystem and the coprocessor. The memory subsystem consists of separate instruction and data memories (L1 cache), a unified L2 memory, and a main memory. The coprocessor consists of three pipeline stages: EMIF_1, CoProc, and EMIF_2. The coprocessor uses it local memory for computations. The data transfer between coprocessor local memory and the main memory is handled by the DMA controller shown in Figure 2.2. Similarly, Figure 2.3 shows the pipeline level description of the DLX processor shown in Figure 2.2. The DLX processor has five pipeline stages: fetch (IF), decode (ID), execute (EX), memory (MEM), and write back (WB).

The architecture shown in Figure 2.2 can issue up to two operations (an ALU or memory access operation and a coprocessor operation) per cycle. The coprocessor supports vector arithmetic operations. This section describes how to specify processor, coprocessor, and memory architectures using the EXPRESSION ADL. It also briefly describes how to capture interrupts and exceptions in the ADL.

2.2.1 Processor Specification

This section describes how EXPRESSION ADL captures the structure and behavior of the DLX processor shown in Figure 2.3.

Structure

The structure of a processor can be viewed as a net-list with the components as nodes and the connectivity as edges. Figure 2.4(a) shows a portion of the EXPRESSION description of the processor. It describes all the components in the structure: *PC*, *Registers*, *fetch*, *decode*, *ALU*, *MEM*, and *writeback*. Each component has a list of attributes. For example, the *ALU* unit has information regarding the number of instructions executed per cycle, timing of each instruction, supported opcodes, input/output latches, and so on.

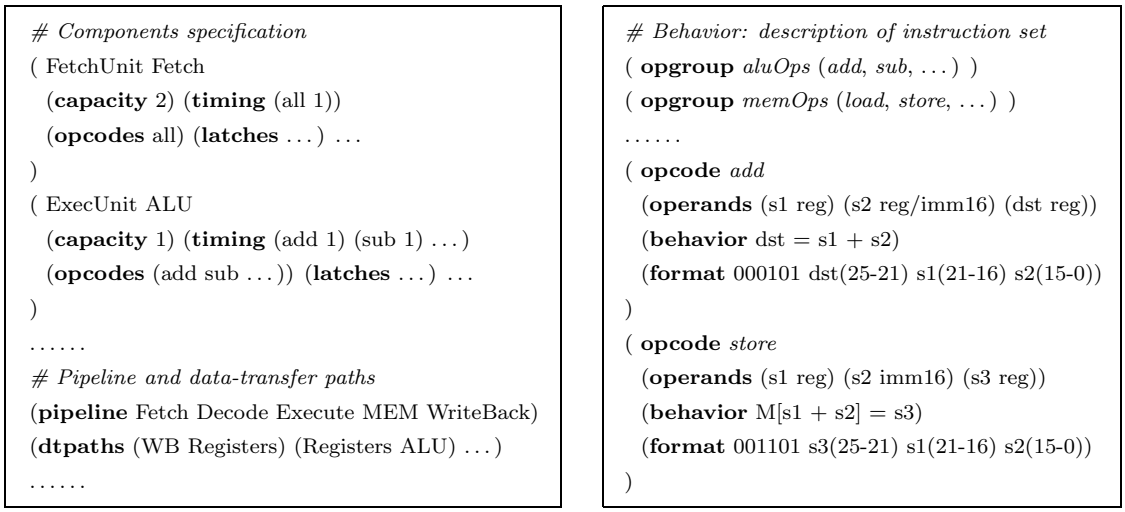
The connectivity is established using the description of pipeline and data-transfer paths. Informally, a pipeline path is used to transfer instruction whereas a data-transfer path is used to transfer data. For example, $\{IF \rightarrow ID \rightarrow EX \rightarrow MEM \rightarrow WB\}$ is a pipeline path, and $\{WB \rightarrow Registers\}$ is a data-transfer path in Figure 2.3. Section 3.1.1 defines the pipeline and data-transfer paths in detail.

Figure 2.4(a) describes the five-stage pipeline as {*fetch, decode, execute, memory, writeback*}. In this particular case, the execute stage has only one component. In general, the execute stage can have multiple execution paths. Furthermore, each path can contain pipelined or multi-cycle execution units. The ADL specification also includes the description of all the data-transfer paths.

Behavior

The EXPRESSION ADL captures the behavior of the architecture as the description of the instruction set. The behavior is organized into operation groups, with each group containing a set of operations[2] having some common characteristics. For example, Figure 2.4(b) shows two operation groups. The *aluOps* includes all the operations supported by the *ALU* unit. Similarly, the *memOps* group contains all the

[2]In this thesis we use the terms operation and instruction interchangeably.



(a) Structure (b) Behavior

Figure 2.4: Processor specification using EXPRESSION ADL

operations supported by the *MEM* unit. Each instruction is then described in terms of its 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 instruction in both binary and assembly. Figure 2.4(b) shows the description of the *add* and *store* operations.

The ADL also captures the mapping between the structure and the behavior (and vice versa). For example, the *add* and *sub* instructions are mapped to the ALU unit, the *load* and *store* instructions are mapped to the *MEM* unit, and so on.

2.2.2 Coprocessor Specification

The ADL specification of a programmable coprocessor is similar to the specification of the processor architecture described in Section 2.2.1. This section describes how the ADL captures the structure and behavior of the coprocessor shown in Figure 2.2(b). To describe the structure of the coprocessor we specify each pipeline stage of the coprocessor along with the processor pipeline as shown in Figure 2.5(a). The

coprocessor pipeline has three stages. The *EMIF_1* (external memory interface) stage requests the DMA to transfer the data from the main memory to the coprocessor local memory. The *CoProc* stage performs the intended computation using the coprocessor local memory for accessing input operands. Results are stored back in the coprocessor memory. Finally, the *EMIF_2* requests the DMA to transfer the data from coprocessor memory to main memory. Figure 2.5(a) shows the description of the *CoProc* component. It supports four-cycle vector arithmetic operations.

```
# Components specification
( FetchUnit Fetch
  (capacity 2) (timing (all 1))
  (opcodes all) (latches ...) ...
)
( CPunit CoProc
  (capacity 1) (timing (vectAdd 4) (vectMul 4))
  (opcodes (vectAdd vectMul ...)) ...
)
# Pipeline and data-transfer paths
(pipeline Fetch Decode Execute MEM WriteBack)
(Execute (parallel ALU Coprocessor))
(Coprocessor (pipeline EMIF_1 CoProc EMIF_2))
(dtpaths (EMIF_1 DMA) (EMIF_2 DMA) ...)
......
```

(a) Structure

```
# Behavior: description of instruction set
( opgroup cpOps
  (vectAdd, vectMul, ...)
)
......
( opcode add
  (operands (s1 reg) (s2 reg/imm16) (dst reg))
  (behavior dst = s1 + s2)
  (format 000101 dst(25-21) s1(21-16) s2(15-0))
)
( opcode vectMul
  (operands (s1 mem) (s2 mem) (dst mem)
            (length imm) )
  (behavior dst = s1 * s2)
)
```

(b) Behavior

Figure 2.5: Coprocessor specification using EXPRESSION ADL

The behavior of the coprocessor is captured in terms of the operations it supports. For example, Figure 2.5(b) shows the description of a *vectMul* operation. Unlike normal instructions whose source and destination operands are of type register (except load/store), here source and destination operands are of type memory. The *s1* and *s2* fields refer to the starting addresses of two source operands for the multiplication. Similarly *dst* refers to the starting address of the destination operand. The *length* field refers to the vector length of the operation that has immediate data type.

2.2.3 Memory Subsystem Specification

In order to explicitly describe the memory architecture in EXPRESSION, we need to capture both structure and behavior of the memory subsystem. The memory structure refers to the organization of the memory subsystem containing memory modules and the connectivity among them. The behavior refers to the memory subsystem instruction set.

```
# Storage section
( DCache L1Data
  (wordsize 64) (linesize 8) (associativity 2)
  (num_lines 1024) (replacement LRU) (latency 1) ...
)
( ICache L1Inst (latency 1) ...)
( DCache L2 (latency 5) ...)
( DRAM MainMemory (latency 50) ...)
# Pipeline and data-transfer paths
(dtpaths (L1Data L2) (L1Inst L2) (L2 MainMemory) ...)
```

(a) Structure

```
# Behavior: description of instruction set
( opcode load_miss
  (operands (s1 L2) (dst L1Data))
  (behavior dst = s1)
)
......
```

(b) Behavior

Figure 2.6: Memory subsystem specification using EXPRESSION ADL

The memory subsystem structure is represented as a netlist of memory components connected through ports and connections. The memory components are described and attributed with their characteristics (such as cache line size, replacement policy, and write policy). For example, Figure 2.6(a) shows the structure of the memory subsystem shown in Figure 2.2(a). The specification of the memory structure also includes the description of the memory pipeline and data-transfer paths. The memory subsystem instruction set represents the possible operations that can occur in the memory subsystem, such as data transfers between different memory modules or to the processor (e.g., load and store), control instructions for the different memory components (such as the DMA), or explicit cache control instructions (e.g., cache freeze, prefetch, replace and refill). For example, Figure 2.6(b) shows an internal memory data transfer operation during a load miss. The *load_miss* operation represents data refill from L2 cache in the event of a L1 data miss.

2.2.4 Specification of Interrupts and Exceptions

It is also necessary to capture exceptions and interrupts explicitly in the ADL for various reasons. First, the simulator and hardware generators require this information to accurately generate and handle exceptions. Second, the specification validation techniques use this information to analyze pipeline interactions in the presence of multiple exceptions. For example, we have used this information in Section 3.2 to verify in-order execution of pipelined processor specifications.

We classify exceptions into three categories: opcode related exceptions, exceptions related to functional units, and external exceptions. The motivation behind this classification is to enable ease of specification.

Opcode related exceptions

It is appropriate to describe opcode related exceptions and their actions inside the opcode specification. For example, the modified *div* operation contains the exception information as shown in Figure 2.7.

```
# Behavior: description of instruction set
......
( opcode div
  (operands (s1 reg) (s2 reg) (dst reg)) (behavior dst = s1 / s2) ...
  (exceptions (if (s2 == 0) throw div_by_zero) ... )
)
```

Figure 2.7: Specification of division_by_zero exception

Exceptions Related to Functional Units

Functional unit related exceptions are defined in ADL's component specification section. For example, the *Decode* unit shown in Figure 2.2 can issue upto two instructions per cycle. The first one is for the ALU pipeline and the second one is for the coprocessor pipeline. It is an exception if the second instruction is not a coprocessor

instruction. The specification of such an exception is described in the *Decode* unit as shown in Figure 2.8.

```
# Components specification
......
( DecodeUnit Decode
  (capacity 2) (timing (all 1)) (opcodes all) ...
  (exceptions (if (slot2 opcode != coprocessor_type) throw illegal_slot_instruction) ...)
)
```

Figure 2.8: Specification of illegal_slot_instruction exception

External Exceptions

External interrupts can be specified at the processor level. We model a control unit that performs the task of a controller. The control unit is also used to perform stalling and flushing of the processor pipelines as described in Section 4.2.4. We describe external interrupts in the control unit. For example, a machine reset exception can be described in the control unit as shown in Figure 2.9. We assume that the *reset* is an external interrupt that is used to generate the internal exception *machine_reset*.

```
# Components specification
......
( ControlUnit control
  ......
  (exceptions (if reset throw machine_reset) ...)
)
```

Figure 2.9: Specification of machine_reset exception

The mapping between exceptions and interrupts is a *many-to-one* mapping function. A class of exceptions may give rise to one interrupt, in that case the architecture implementation should ensure that only one exception from that class occurs at a time. In general, one interrupt corresponds to more than one exception. We specify the interrupts and exceptions in the control unit specification. For example, the interrupt *int1* is described in Figure 2.10. The interrupt *int1* gets generated due

to any memory failure during memory operation, for example, ITLB miss or DTLB miss. It can mask several lower priority interrupts such as *int2* and *int7*.

```
# Components specification
.....
( ControlUnit control
  .....
  (interrupt int1
   (exceptions ITLB_miss DTLB_miss ...)
   (masks int2 int7 ...) (behavior ...) ...
  )
)
```

Figure 2.10: Specification of interrupts

We model the interrupt handler using a priority table that can accept n exception requests and generate only one interrupt per cycle. The multiple exceptions are handled in a simple and uniform manner using interrupt service register (ISR). The length of the ISR is equal to the number of interrupts possible in that architecture. One entry in the ISR corresponds to an interrupt. Control unit defines the class of exceptions that generates a particular interrupt. Each exception sets one particular bit in the ISR of the interrupt handler. Interrupt handler decides the highest priority interrupt using the interrupt priority table. Depending on the masking information the highest priority interrupt masks the appropriate bits in ISR. The process of selecting highest priority interrupt continues until there are no bits set in ISR. The details on specification of exceptions and interrupts are available in [58].

2.3 Chapter Summary

This chapter surveyed existing ADLs in terms of their capabilities in capturing programmable architectures. Structural ADLs enable flexible and precise micro-architecture descriptions. The same description can be used for hardware synthesis, test generation, simulation and compilation. However, it is difficult to extract

instruction-set information for retargetable compilation. Behavioral ADLs simplify the instruction set description by exploiting the common components between operations. However, the lack of detailed pipeline and timing information prevents the use of these languages as an extensible architecture model. Mixed ADLs capture both the structure and the behavior of the architecture.

The second part of this chapter described the use of the EXPRESSION ADL in our framework to specify programmable embedded systems. We described how to capture processor, coprocessor, and memory architectures using the ADL. Chapter 3 will present techniques to validate the ADL specification of the architecture.

Chapter 3

Validation of Specification

One of the most important requirements in a top-down validation methodology is to ensure that the specification (reference model) is golden. This chapter presents techniques to validate the static and dynamic behaviors of the architecture specified in an ADL. It is necessary to validate the ADL specification to ensure the correctness of both the architecture specified and the generated executable models including software toolkit and hardware implementation. The benefits of validation are two-fold. First, the process of any specification is error-prone and thus verification techniques can be used to check for correctness and consistency of the specification. Second, changes made to the processor during exploration may result in incorrect execution of the system and verification techniques can be used to ensure correctness of the modified architecture.

One of the major challenges in validating the ADL specification is to verify the pipeline behavior in the presence of hazards and multiple exceptions. There are many important properties that need to be verified to validate the pipeline behavior. For example, it is necessary to verify that each operation in the instruction set can execute correctly in the processor pipeline. It is also necessary to ensure that execution of each operation is completed in a finite amount of time. Similarly, it is important to verify the execution style (e.g., in-order execution) of the architecture. In this chapter, we verify several properties to validate the static and dynamic behaviors of the specified pipelined architecture. While these properties are by no means complete to prove

the correctness of the specification, we believe these are necessary for establishing the correctness of the specification. Additional properties can easily be added and integrated into our validation framework.

The chapter is organized as follows. Section 3.1 describes the validation techniques to ensure that the static behavior of the pipeline is well-formed by analyzing the structural aspects of the specification using a graph based model. Section 3.2 presents the techniques to verify the dynamic behavior by analyzing the instruction flow in the pipeline using a FSM based model to validate several important architectural properties such as determinism and in-order execution in the presence of hazards and multiple exceptions. Finally, Section 3.3 summarizes the chapter.

3.1 Validation of Static Behavior

This section presents an automatic validation framework driven by an ADL. The first step (and only manual step) in the flow is to specify the architecture using EXPRESSION ADL. A novel feature of this approach is the ability to model the pipeline structure and behavior of the processor, co-processor, and memory subsystem using a graph-based model. Based on this model, we present algorithms to ensure that the static behavior of the pipeline is well-formed by analyzing the structural aspects of the specification.

Figure 3.1 shows the flow for validating static behaviors. The designer describes the programmable architecture in an ADL. The graph model of the architecture is generated from this ADL description. Several properties are applied to ensure that the architecture is well formed.

This section describes three important steps in this methodology. First, it presents a graph-based modeling of processor, memory, and co-processor pipelines. Second, it describes several properties that must be satisfied for valid pipeline specification. Finally, it illustrates validation of pipeline specifications for several realistic architectures.

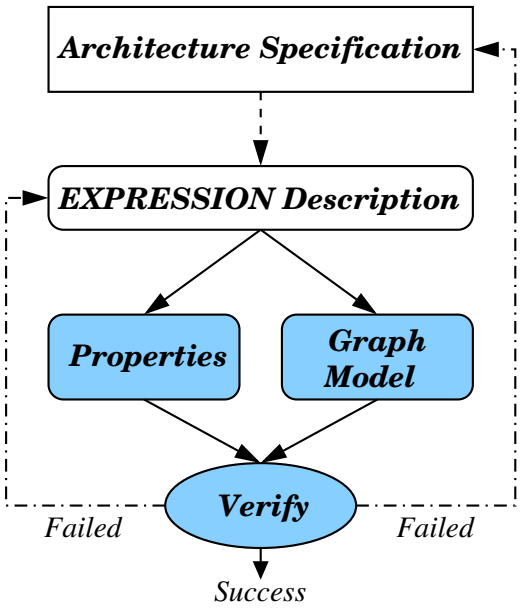


Figure 3.1: Validation of pipeline specifications

3.1.1 Graph-based Modeling of Pipelines

We present a graph-based modeling of architecture pipelines that captures both the structure and the behavior. The graph model presented here can be derived from a pipeline specification of the architecture described in an ADL e.g., EXPRESSION [20]. This graph model can capture processor, memory, and co-processor pipelines for a wide variety of architectures including RISC, DSP, VLIW, and superscalar architectures. 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.

Structure

The structure of an architecture pipeline is based on a block diagram view available in the processor manual, and is modeled as a graph $G_S = (V_S, E_S)$, where V_S denotes a set of components and E_S consists of a set of edges. V_S consists of two types of components: V_{unit} and $V_{storage}$. V_{unit} is a set of *functional units* (e.g., ALU), and $V_{storage}$ is a set of *storages* (e.g., register files). E_S consists of two types of edges.

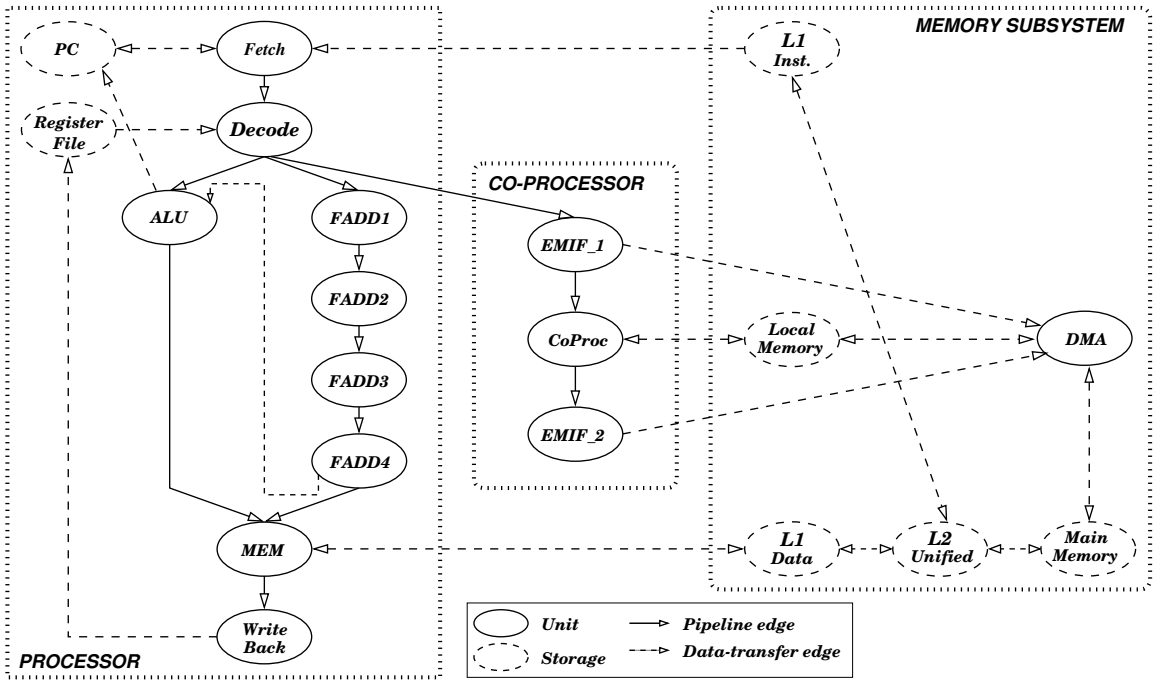


Figure 3.2: An example architecture

$E_{data\_transfer}$ is a set of *data-transfer edges*, and $E_{pipeline}$ is a set of *pipeline edges*. An edge (pipeline or data-transfer) indicates connectivity between two components. A data-transfer edge transfers data between units and storages. A pipeline edge transfers instruction (operation) between two units.

$$
\begin{aligned}
V_S &= V_{unit} \cup V_{storage} \\
E_S &= E_{data\_transfer} \cup E_{pipeline} \\
E_{data\_transfer} &\subseteq \{V_{unit},\ V_{storage}\} \times \{V_{unit},\ V_{storage}\} \\
E_{pipeline} &\subseteq V_{unit} \times V_{unit}
\end{aligned}
$$

For illustration, we use a simple multi-issue architecture consisting of a processor, a co-processor and a memory subsystem. Figure 3.2 shows the graph-based model of this architecture that can issue up to three operations (an ALU operation, a floating-point addition operation, and a coprocessor operation) per cycle. Figure 3.2 is obtained from Figure 2.2. We have added a four-stage floating point adder (*FADD*) and a feedback path from the *FADD* pipeline to the *ALU* pipeline. In the figure,

oval boxes denote units, dotted ovals are storages, bold edges are pipeline edges, and dotted edges are data-transfer edges. A path from a root node (e.g., Fetch) to a leaf node (e.g, WriteBack) consisting of units and pipeline edges is called a *pipeline path*. For example, one of the pipeline paths is {*Fetch, Decode, ALU, MEM, Write-Back*}. 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, {*MEM, L1, L2, MainMemory*} is a data-transfer path.

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 that can be executed on the architecture. Each operation in turn consists of a set of fields (e.g. opcode, arguments) that specify, at an abstract level, the execution semantics of the operation. We model the behavior as a graph $G_B = (V_B, E_B)$, where V_B is a set of nodes, and E_B is a set of edges. The nodes represent the fields of each operation, while the edges represent orderings between the fields. The behavior graph G_B is a set of disjointed sub-graphs, and each sub-graph is called an *operation graph* (or simply an operation). Figure 3.3 shows a portion of the behavior (consisting of two operation graphs) for the example processor shown in Figure 3.2.

$$
\begin{aligned}
V_B &= V_{opcode} \cup V_{argument} \\
E_B &= E_{operation} \cup E_{execution} \\
E_{operation} &\subseteq V_{opcode} \times V_{argument} \quad \cup \quad V_{argument} \times V_{argument} \\
E_{execution} &\subseteq V_{argument} \times V_{argument} \quad \cup \quad V_{argument} \times V_{opcode}
\end{aligned}
$$

Nodes are of two types. V_{opcode} is a set of opcode nodes that represent the opcode (i.e. mnemonic), and $V_{argument}$ is a set of argument nodes that represent argument fields (i.e., source and destination arguments). In Figure 3.3, the ADD and STORE nodes are opcode nodes, while the others are argument nodes. Edges are also of two types. $E_{operation}$ is a set of operation edges that link the fields of the operation and also specify the syntactical ordering between them. On the other hand, $E_{execution}$ is

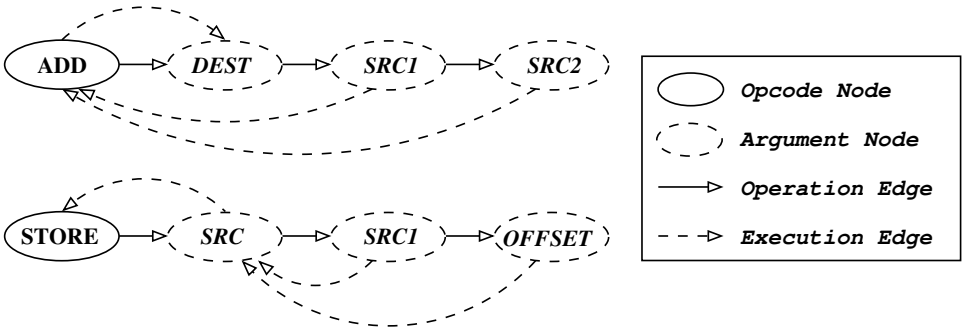


Figure 3.3: A fragment of the behavior graph

a set of execution edges that specify the execution ordering between the fields. In Figure 3.3, the solid edges represent operation edges while the dotted edges represent execution edges. For the ADD operation, the operation edges specify that the syntactical ordering is opcode followed by DEST, SRC1 and SRC2 arguments (in that order), and the execution edges specify that the SRC1 and SRC2 arguments are executed (i.e., read) before the ADD operation is performed. Finally, the DEST argument is written.

Mapping between Structure and Behavior

The mapping between the structure and the behavior is captured explicitly in the ADL. This information is available in the architecture manual as mapping between the instruction set and the functional units. It other words, the manual describes what operations are supported by which functional units in the architecture. We define a set of mapping functions that map nodes in the structure to nodes in the behavior (and vice-versa). The *unit-to-opcode (opcode-to-unit)* mapping is a bi-directional function that maps unit nodes in the structure to opcode nodes in the behavior. The *unit-to-opcode* mappings for the architecture in Figure 3.2 include mappings from *Fetch* unit to opcodes {*ADD, FADD*}, *ALU* unit to opcode *ADD*, *FADD1* unit to opcode *FADD* 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-storage* mappings for the *ADD* operation are mappings from {*DEST, SRC1, SRC2*} to *RegisterFile*.

Each functional unit (with read or write ports) supports certain data-transfer operations. These operations can be derived from the above mapping functions. For example, the *Decode* unit of Figure 3.2 supports register read (*regRead*) for ADD and LD opcodes; the *MEM* unit supports data read (*dataRead*) and data write (*dataWrite*) from L1 data cache; the *Fetch* unit supports instruction read (*instRead*) from L1 instruction cache; the *WriteBack* unit supports register write (*regWrite*). Similarly, each storage supports certain data-transfer operations. For example, the *RegisterFile* of Figure 3.2 supports *regRead* and *regWrite*; L1 data cache supports *dataRead* and *dataWrite*, and so on.

3.1.2 Validation of Pipeline Specifications

Based on the graph model presented in the previous section, the ADL specification of architecture pipelines can be validated. In this section, we describe some of the properties used in our framework for validating pipelined architecture specifications.

Connectedness Property

The connectedness property ensures that each component is connected to other component(s). As pipeline and data-transfer paths are connected regions of the architecture, this property holds if each component belongs to at least one pipeline or data-transfer path.

$$\forall v_{comp} \in V_S, (\exists G_{PP} \in \mathbf{G_{PP}},\ s.t.\ v_{comp} \in G_{PP}) \vee (\exists G_{DP} \in \mathbf{G_{DP}},\ s.t.\ v_{comp} \in G_{DP})$$

where $\mathbf{G_{PP}}$ is a set of pipeline paths and $\mathbf{G_{DP}}$ is a set of data-transfer paths.

Algorithm 1 presents the pseudo-code for verifying the connectedness property. The algorithm requires the graph model G of the architecture as input. It also requires all the component lists as input. The first step is to unmark the entries in all the input lists. Each input list contains all the respective components in the graph. For example, the *ListOfUnits* contains all the units in the graph G. Next, the graph is traversed in breadth-first manner and the visited components are marked. For example, when a unit u is visited during traversal, it is marked in *ListOfUnits*.

Finally, the algorithm returns true if all the entries are marked in all the input lists. It returns false if there are any unmarked entries in any of the input lists, and it reports them.

Algorithm 1: *Verify Connectedness*
Inputs: i. Graph model of the architecture G
ii. *ListOfUnits*: list of units in the graph G
iii. *ListOfStorages*: list of storages in the graph G
Outputs: i. *True*, if the graph model satisfies this property else *false*.
ii. In case of failure, report the disconnected components.
Begin
Unmark all the entries in all the input lists.
InsertQ(root, Q) /* Put root node of G in queue Q */
while Q is not empty
Node n = DeleteQ(Q) /* Remove the front element of Q */
Mark n as visited in G
case type of node n
unit: Mark n in *ListOfUnits*
storage: Mark n in *ListOfStorages*
endcase
for each successor node s of n
if s is not visited InsertQ(s, Q)
endfor
endwhile
Return *true* if all the entries are marked in all of the input lists;
false otherwise, and report the unmarked components.
End

Each node of the graph is visited only once. The time and space complexity of the algorithm is $O(n)$, where n is the number of nodes in the graph G. Each node of the graph can be either unit or storage.

False Pipeline and Data-transfer Paths

According to the definition of pipeline paths, there may exist pipeline paths that are never activated by any operation. Such pipeline paths are said to be *false*. For

example, let us use another architecture shown in Figure 3.4 that executes two operations: ALU-shift (*alus*) and multiply-accumulate (*mac*). This processor has unit-to-opcode mappings between ALU unit and opcode *alus*, between SFT and *alus*, between MUL and *mac*, and between ACC and *mac*. Also, there are unit-to-opcode mappings between each of {*IFD, RD1, RD2, WB*} and *alus*, and each of {*IFD, RD1, RD2, WB*} and *mac*. This processor has four pipeline paths: {*IFD, RD1, ALU, RD2, SFT, WB*}, {*IFD, RD1, MUL, RD2, ACC, WB*}, {*IFD, RD1, ALU, RD2, ACC, WB*}, and {*IFD, RD1, MUL, RD2, SFT, WB*}. However, the last two pipeline paths cannot be activated by any operation. Therefore, they are false pipeline paths. Since these false pipeline paths may become false paths depending on the detailed structure of RD2, they should be detected at a higher level of abstraction to ease the later design phases.

From the view point of SOC architecture exploration, we can view the false pipeline paths as an indication of potential behaviors that are not explicitly defined in the ADL description. These false pipeline paths can be used to perform valid computations. This opens up avenues for further exploration experiments for cost, power, and performance by adding new instructions to activate the false pipeline paths.

Formally, a pipeline path $G_{PP}(V_{PP}, E_{PP})$ is false if the intersection of opcodes supported by the units in the pipeline path is empty.

$$\bigcap_{v_{unit} \in V_{PP}} f_{unit-opcode}(v_{unit}) = \phi \tag{3.1}$$

Similarly, there may exist data-transfer paths in the specification that are never activated by any operation. Such data-transfer paths are said to be *false*. For example, let us use another architecture shown in Figure 3.5 that has seven possible data-transfer operations: integer-register-read (IregRd), float-register-read (FregRd), integer-register-write (IregWr), float-register-write (FregWr), load-data-from-memory (ldData), load-instruction-from-memory (ldInst), and store-data-in-memory (stData). The *Decode (ID)* unit has mappings for *IregRd* and *FregRd*. There are mappings between each of {*WB1, WB2*} and {*IregWr, FregWr*}, each of {*IF, L1I, ISB* } and *ldInst*, each of { *LDST, L1D, DSB* } and { *ldData, stData* }, and each of { *L2, DRAM* } and { *ldData, stData, ldInst* }. This processor has ten data-transfer

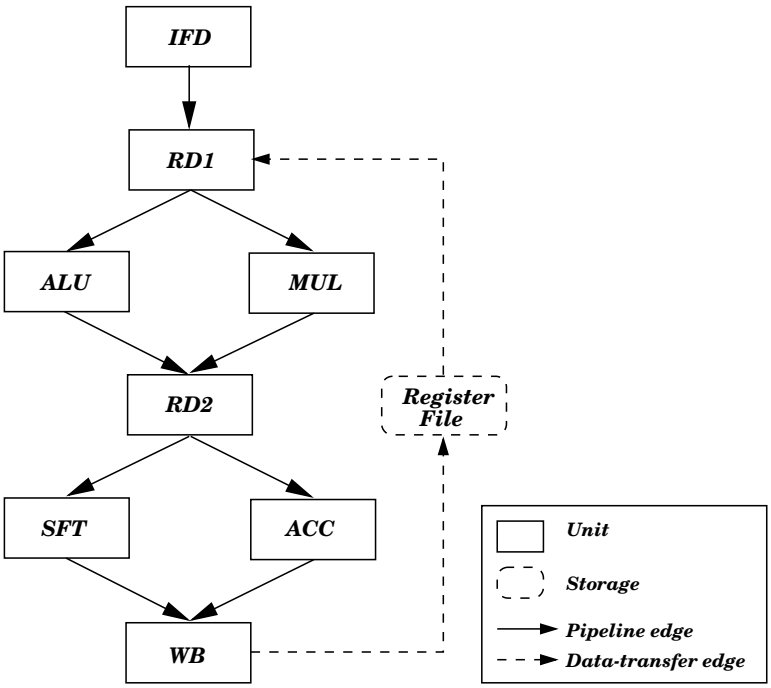


Figure 3.4: An example processor with false pipeline paths

paths: {*IRF, ID*}, {*FRF, ID*}, {*WB1, IRF*}, {*WB1, FRF*}, {*WB2, IRF*}, {*WB2, FRF*}, {*IF, L1I, L2, ISB, DRAM*}, {*LDST, L1D, L2, DSB, DRAM*}, {*IF, L1I, L2, DSB, DRAM*}, {*LDST, L1D, L2, ISB, DRAM*} . However, the last two data-transfer paths cannot be activated by any operation. Therefore, they are false data-transfer paths. If *ALU1* supports only floating-point operations, the fourth path ({*WB1, IRF*}) becomes a false data-transfer path.

Formally, a data-transfer path $G_{DP}(V_{DP}, E_{DP})$ is false if the intersection of data-transfer operations supported by the units and storages ($f_{node-operation}$) in the data-transfer path is empty.

$$\bigcap_{v_{node} \in V_{DP}} f_{node-operation}(v_{node}) = \phi \tag{3.2}$$

Algorithm 2 presents the pseudo-code for verifying false pipeline and data-transfer paths. The algorithm requires the graph model G as input. It traverses the graph in depth-first manner along each pipeline and data-transfer path. Each unit u has a list of supported opcodes $SuppOpList_u$. Each node n (unit or storage) also maintains

Algorithm 2: *Verify False Pipeline and Data-transfer Paths*

Input: Graph model of the architecture G.

Outputs: i. *True*, if the graph model satisfies this property else *false*.

ii. In case of failure, report the list of false pipeline and data-transfer paths.

Begin

Push (root, S); FalsePPpathList = {}; FalseDPpathList = {};

while S is not empty

Node n = Pop(S); Mark n as *visited*.

case node type of n

unit: **if** n is the root node

$OutOpList_n = SuppOpList_n$ // Send supported opcodes to children

else /* p is the recently visited parent */

$InOpList_n = OutOpList_p$; $OutOpList_n = SuppOpList_n \cap InOpList_n$

if n has *read* or *write* ports

$OutDTopList_n = ComputeDataTransferOps(OutOpList_n)$

if $OutDTopList_n$ is empty

for all the data-transfer paths *fDP* from n to any leaf nodes

Insert *fDP* in *FalseDPpathList*.

else for each children storage node *st* of n, Push(st, S)

if $OutOpList_n$ is empty

Get path *pp* from n by tracing recently visited parents till root

for all pipeline paths *ppEnd* from n to any leaf nodes

Append *ppEnd* to *pp* to get *fPP*; Insert *fPP* in *FalsePPpathList*.

else for each children unit u of n, Push(u, S)

storage: $InDTopList_n = OutDTopList_p$

$OutDTopList_n = SuppDTopList_n \cap InDTopList_n$

if $OutDTopList_n$ is empty

Get path *dp* from n by tracing recently visited parents till any unit

for all data-transfer paths *dpEnd* from n to any leaf nodes

Append *dpEnd* to *dp* to generate false data-transfer path *fDP*.

Insert *fDP* in *FalseDPpathList*.

endfor

else for each children storage node *st* of n, Push(st, S)

endcase

endwhile

if *FalsePPpathList* **and** *FalseDPpathList* are empty **return** *true*;

else return *false* and report *FalsePPpathList* and *FalseDPpathList*.

End

four temporary lists: $OutOpList_n$, $OutDTopList_n$, $InOpList_n$, and $InDTopList_n$. The $OutOpList_n$ is the list of opcodes produced by unit n and sent to its children units. The $OutDTopList_n$ is the list of data-transfer operations produced by node (unit or storage) n and sent to its children storages. The $InOpList_n$ is the list that is used by unit n to copy the $OutOpList_p$, the output list of the recently visited parent p. Similarly, the $InDTopList_n$ is the list that is used by storage n to copy the $OutDTopList_p$, the output list of the recently visited parent p. Each unit n performs intersection of $InOpList_n$ and $SuppOpList_n$ and send the result $OutOpList_n$ to its children units. If $OutOpList_n$ is empty, all the pipeline paths that use the path from n to root (via recently visited parents) are false pipeline paths. A unit with read or write ports computes data transfer operations using the method described in Section 3.1.1. A storage computes $OutDTopList_n$ by performing intersection of $SuppDTopList_n$ and the input list $InDTopList_n$. If $OutDTopList_n$ is empty, all the data-transfer paths that use the path from storage n to any unit via recently visited parents are false data-transfer paths. The algorithm returns true if there are no false pipeline or data-transfer paths. It returns false if there are any false pipeline or data-transfer paths, and reports them.

If there are n nodes, x pipeline and data-transfer paths in the graph and the number of opcodes supported by the processor is p, the time complexity of the algorithm is $O(x \times n \times (x + plogp))$ and space complexity is $O(n \times p)$. The supported opcode list in each node is a sorted list.

Completeness Property

The completeness property confirms that all operations must be executable. An operation *op* is executable if there exists a pipeline path $G_{PP}(V_{PP}, E_{PP})$ on which *op* is executable. An operation *op* is executable on a pipeline path $G_{PP}(V_{PP}, E_{PP})$ if both Condition 3.1 and 3.2 hold.

Condition 3.1: All units in V_{PP} support the operation *op*. More formally, the following condition holds where v_{opcode} is the opcode of the operation *op*.

$$\forall v_{unit} \in V_{PP}, v_{opcode} \in f_{unit-opcode}(v_{unit}). \tag{3.3}$$

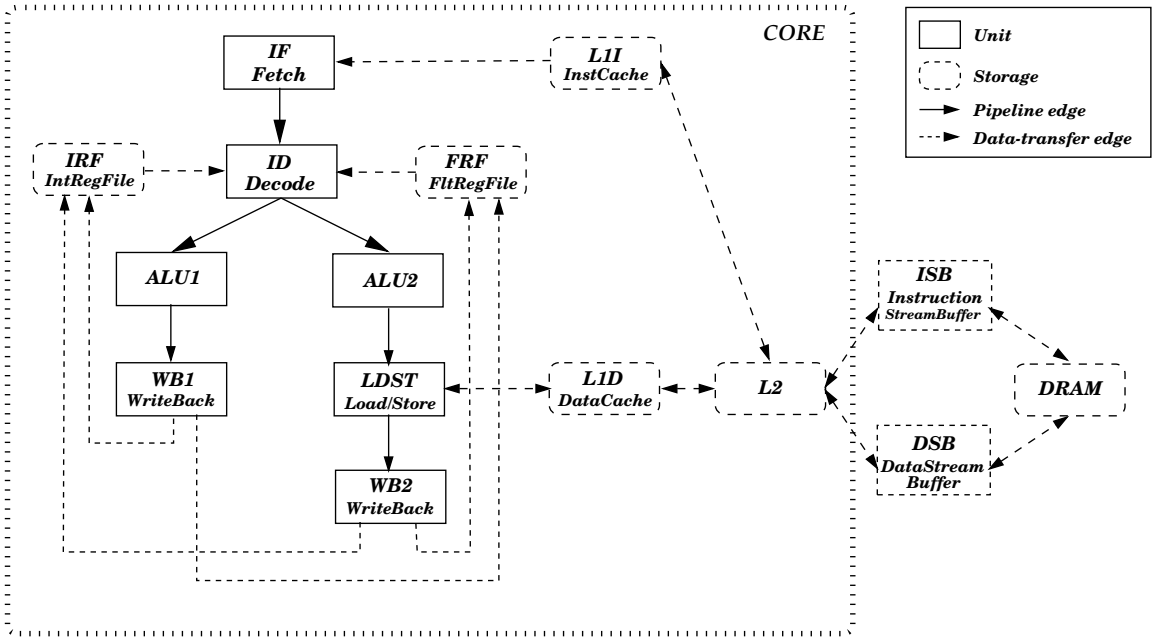


Figure 3.5: An example processor with false data-transfer paths

Condition 3.2: There are no conflicting partial orderings of operation arguments and unit ports. Let V be a set of argument nodes of operation *op*. There are no conflicting partial orderings of operation arguments and unit ports if, for any two nodes $v_1, v_2 \in V$ such that $(v_1, v_2) \in E_{execution}$, all the following conditions hold:

- There exists a data-transfer path from a storage $f_{arg-storage}(v_1)$ to a unit v_{u1} in V_{PP} through a port $f_{arg-port}(v_1)$.
- There exists a data-transfer path from a unit v_{u2} in V_{PP} to a storage $f_{arg-storage}(v_2)$ through a port $f_{arg-port}(v_2)$.
- v_{u1} and v_{u2} are the same unit or there is a path consisting of pipeline edges from v_{u1} to v_{u2}.

For example, let us consider the *ADD* operation (shown in Figure 3.3) for the processor described in Figure 3.2. To satisfy Condition 3.1, each of {*Fetch, Decode, ALU, MEM, WriteBack*} must have mappings to the *ADD* opcode. On the other hand, Condition 3.2 is satisfied because the structure has data-transfer paths from

Algorithm 3: *Verify Completeness*

Inputs: i. Graph model G of the architecture.

ii. The list of operations *OpList* supported by the architecture.

Outputs: i. *True*, if the graph model satisfies this property else *false*.

ii. In case of failure, report the list of operations that are not executable.

Begin

for each operation *op* supported by the architecture /* $op \in OpList$ */

opSrcList = list of sources in the operation *op*.

opDestList = list of destinations in the operation *op*.

Push(root, S) /* Put root node of G in stack S */

while S is not empty

Node n = Pop(S); Mark n as *visited* in G.

if $op \in SuppOpList_n$ /* *op* is supported by unit n */

for each port p of n

if p is a read or read-write port

for each unmarked source *src* in *opSrcList*

if *src* can be read via p, mark *src* in *opSrcList* with *(p, n)*

if p is a write or read-write port

for each unmarked destination *dest* in *opDestList*

if *dest* can be written via p

mark *dest* in *opDestList* with *(p, n)*

endfor

if unit n is a leaf node

if ((all sources in *opSrcList* are marked) **and**

(all nodes r that read the sources are in expected order) **and**

(all destinations in *opDestList* are marked) **and**

(all nodes w that write the destinations are in expected order) **and**

(all nodes r & w are in same pipeline path and r appears before w))

Mark *op* in *OpList* /* this path supports *op* */

break /* one pipeline path is sufficient, exit while loop */

endif

else for each children unit u of n, Push(u, S)

endwhile

endfor

Return *true* if all the entries in *OpList* are marked;

false otherwise, and report the unmarked entries in *OpList*.

End

RegisterFile to *Decode* and from *WriteBack* to *RegisterFile*, and there is a pipeline path from *Decode* to *WriteBack*.

Algorithm 3 presents the pseudo-code for verifying the completeness property. The algorithm requires the graph model (G) and the list of operations supported by the architecture (*OpList*) as inputs. It traverses the graph in depth-first manner for each operation *op* and identifies a pipeline path *pp* that supports *op*. All the units n in the pipeline path should have *op* in their supported opcode list $SuppOpList_n$. The pipeline path *pp* must have units that can read the source operands of *op* and write the destination operands of *op* in correct order. If all the conditions are met, *op* is executable in pipeline path *pp* and *op* is marked in *OpList*. The algorithm returns true if all the entries in *OpList* are marked. It returns false if there are unmarked entries and reports them.

If there are n nodes, x pipeline and data-transfer paths in the graph and the number of opcodes supported by the architecture is p, the time complexity of the algorithm is $O(x \times n \times p \times logp)$ and space complexity is $O(n \times p)$. The opcode list in each unit is a sorted list.

Finiteness Property

The finiteness property guarantees the termination of any operation executed through the pipeline. The termination is guaranteed if all pipeline and data-transfer paths except false pipeline and data-transfer paths have finite length and all nodes on the pipeline or data-transfer paths have finite timing. The length of a pipeline or data-transfer path is defined as the number of stages required to reach the final (leaf) nodes from the root node of the graph. Formally,

$$\exists K,\ s.t.\ \forall path \in (\mathbf{G_{PP}}, \mathbf{G_{DP}}), num\_stages(path) < K \tag{3.4}$$

Here, *num_stages* is a function that, given a pipeline or data-transfer path, returns the number of stages (i.e. clock cycles) required to execute it. In the presence of cycles in the pipeline path, this function cannot be determined from the structural graph model alone. However, if there are no cycles in the pipeline paths, the termination property

is satisfied if the number of nodes in V_S is finite, and each multi-cycle component has finite timing.

Algorithm 4: *Verify Finiteness*
Inputs: i. Graph model G of the architecture.
ii. The list of operations *OpList* supported by the architecture.
Outputs: i. *True*, if the graph model satisfies this property else *false*.
ii. In case of failure, report the list of paths that violates this property.
Begin
PathList = {};
for each operation *op* supported by the architecture
PathLength = 0; *ColorCode* = 0
Push(< *root*, *PathLength* >, *S*); Unmark all the nodes in graph *G*
while *S* is not empty
< *n*, *PathLength* > = Pop(*S*)
if *op* ∈ $SuppOpList_n$ /* *op* is supported by unit *n* */
PathLength = *PathLength* + 1; *timing* = GetExecutionTime(*op*, *n*);
if ((*n* is already marked with *ColorCode*) **or**
(*timing* is greater than *MaxExecutionTime*) **or**
(*PathLength* is greater than *MaxPathLength*))
Insert < *op*, *path* > pair in *PathList*; **break**; /* exit while loop */
else
Mark *n* with *ColorCode*
if unit *n* is a leaf node, *ColorCode* = *ColorCode* + 1;
else
for each children node *child* of *n*
Push(< *child*, *PathLength* >, *S*);
endif
endif
else *ColorCode* = *ColorCode* + 1;
endwhile
endfor
Return *true* if *PathList* is empty
false otherwise, and report *PathList*.
End

Algorithm 4 presents the pseudo-code for verifying finiteness property. The algorithm requires the graph model G and the list of operations supported by the architecture (*OpList*) as inputs. It traverses the graph in depth-first manner for each operation *op* and identifies all the pipeline paths *op-pp* that support *op*. For each operation it marks different pipeline paths *op-pp* with a different color. A cycle is detected if the same colored node is visited more than once during traversal. The pipeline path *op-pp* with cycle will be stored in *PathList*. This property is also violated when there are paths that are longer than *MaxPathLength* or when the execution time needed by *op* in any node in that path is greater than *MaxExecutionTime*. The algorithm returns true if *PathList* is empty. It returns false if there are entries in *PathList* and reports them.

Our finiteness algorithm assumes that there are no cycles in the pipeline. If the cycles are allowed in the pipeline due to the reuse of the resources, our algorithm needs to be modified. Let us assume that a resource is reused by an operation *op* for n_{op} times. We can modify the algorithm to check for *"already marked with ColorCode for n_{op} times"* instead of checking *"already marked with ColorCode"* for the operation *op*.

If there are n nodes, x pipeline and data-transfer paths in the graph and the number of opcodes supported by the architecture is p, the time complexity of this algorithm is $O(x \times n \times p \times logp)$ and space complexity is $O(n \times p)$. The opcode list in each unit is a sorted list.

Architecture-specific Properties

The architecture must be well-formed based on the original intent of the architecture model. To verify the validity of this property we need to verify several architectural properties. Here we mention some of the architecture specific properties we verify in our framework.

- The number of operations processed per cycle by a unit can not be smaller than the total number of operations sent by its parents unless the unit has a reservation station. This event (fewer output instructions than the input

instructions) is not an error if that specific unit kills certain operations based on certain conditions e.g., killing no operation (NOP).

- ❒ There should be a path from an execution unit supporting branch opcodes to program counter (PC) or Fetch unit to ensure that PC is modified in case of branch mis-prediction.

- ❒ The instruction template should match the available pipeline bandwidth. However, having instruction template size different than pipeline bandwidth does not always imply an error because a machine with n operations in an instruction and m ($> n$) parallel pipeline paths may have many multicycle units. Similarly, the architecture may have m ($< n$) parallel pipeline paths if it has a reservation station and the instruction fetch timing is large.

- ❒ There must be a path from load/store unit to main memory via storage components to ensure that every memory operation is complete.

- ❒ The address space used by the processor must be equal to the union of address spaces covered by memory subsystem (SRAM, cache hierarchies etc.) to ensure that all the memory accesses can complete.

These are only some of the properties we currently verify in our framework. In this manner, for every architecture with new architectural features we can easily add and verify new properties for those features.

Algorithm 5 shows how we apply these properties in our framework. We first verify finiteness property before applying any other properties in our framework. If there are paths with infinite length and timing, the finiteness algorithm will display the path and exit. Next, we apply the connectedness property followed by the false pipeline and data-transfer path property. The remaining properties can be applied in any order. The worst case time complexity of Algorithm 5 is $O(x \times n \times (x + plogp))$ and space complexity is $O(n \times p)$, where the architecture graph has n nodes, x pipeline and data-transfer paths, and the number of operations supported by the processor is p. Typically, the numeric values of these variables are not large: both n and x are

less than 100, and p is less than 1000. As a result, it requires less than a second to verify an architecture specification as demonstrated in the next section.

Algorithm 5: *Verify Architecture Specification*
Input: Graph model G of the architecture.
Output: *True*, if the graph model satisfies all the properties
else *false*, and report the error.
Begin
status = VerifyFiniteness (G, *G.SupportedOpcodeList*);
if (status == *false*)
Report the paths that violate this property;
return *false*;
endif
status = VerifyConnectedness (G, *G.ListOfUnits*, ...);
if (status == *false*) {
Report the components that are not connected;
return *false*;
endif
status = VerifyFalsePipelineDataTransferPaths(G);
if (status == *false*) {
Report the list of false pipeline and data-transfer paths;
return *false*;
endif
status = VerifyCompleteness (G, *G.SupportedOpcodeList*);
if (status == *false*) {
Report the list of operations that are not executable;
return *false*;
endif
/* Apply other architecture specific properties */
. . . . . .
return *true*;
End

3.1.3 Experiments

In order to demonstrate the applicability and usefulness of our validation approach, we have described a wide range of architectures using the EXPRESSION ADL: MIPS R10K [31], TI C6x [89], PowerPC [30], and DLX [22] that are collectively representative of RISC, DSP, VLIW, and superscalar architectures. Our framework generates the graph model from the ADL specification. We have implemented each property as a function that operates on the graph model. Finally, we have applied these properties on the graph model to verify that the specified architecture is well-formed. Table 3.1 shows the specification validation time for different architectures on a 333 MHz Sun Ultra-II with 128M RAM. This includes the time to generate the graph model from the ADL specification and to apply all the properties on the graph model. The validation time depends on three aspects: number of properties applied, complexity of the structure and the number of operations supported by the architecture. Typically, the validation time is in the order of seconds.

Table 3.1: Specification validation time for different architectures

Architecture	ARM	DLX	TI C6x	PowerPC	MIPS R10K
Validation Time (seconds)	0.2	0.1	0.2	0.3	0.5

In the remainder of this section, we describe our specification validation experiments. First, we describe the validation of the DLX specification in detail. Next, we summarize the incorrect specification errors captured by our framework during design space exploration of different architectures. We have described the architectures using EXPRESSION ADL. We have performed many modifications of the specification. These are some of the typical modifications an architect would like to do during design space exploration. The errors captured by our framework represent the common mistakes made during modification of the architecture specification.

Validation of the DLX specification

Our framework generated the graph model G from the ADL specification of the DLX architecture. Figure 3.6 shows the simplified graph model of the DLX architec-

ture. Figure 3.6 is obtained by adding two execution paths (seven-stage multiplier and a multi-cycle divider) in the processor pipeline shown in Figure 3.2. The oval (unit) and rectangular (storage) boxes represent nodes. The solid (pipeline) and dotted (data-transfer) lines represent edges.

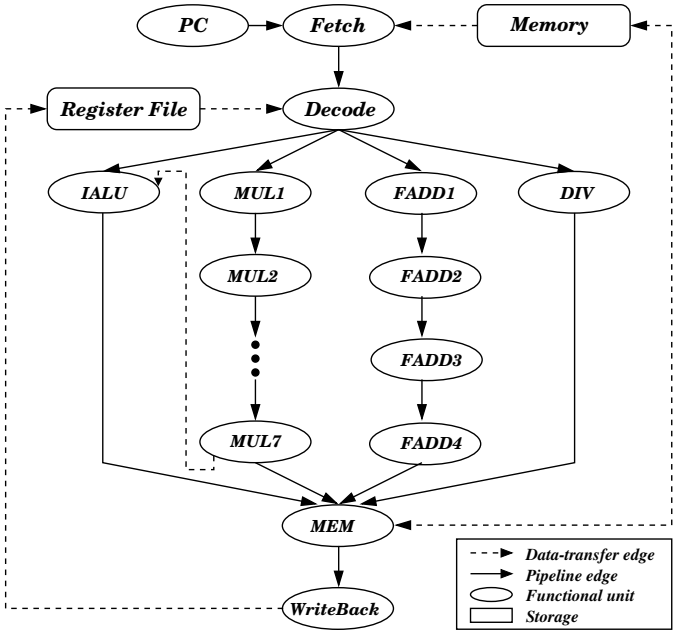


Figure 3.6: The DLX architecture

We applied all the properties (Algorithm 5) on the graph model G. We encountered two kinds of errors viz., incomplete specification errors and incorrect specification errors. An example of an incomplete specification error we uncovered is that the opcode assignment is not done for the fifth stage of the multiplier pipeline. Similarly, an example of an incorrect specification error we found is that only load/store opcodes were mapped for the memory stage (*MEM*). Since all the opcodes pass through memory stage in DLX, it is necessary to map all the opcodes in memory stage as well.

We used Algorithm 5 for specification validation. First, the finiteness property is applied on the graph model. It detects a violation for the division operation since the multi-cycle division unit (*DIV*) has an undefined latency value. Once the latency for the division operation is defined, the finiteness property is successful. Next, the connectedness property is applied. It detects that the sixth stage of the multiplier unit

(*MUL6*) is not connected. Once it is connected properly (from *MUL5* to *MUL6*, and from *MUL6* to *MUL7*), the connectedness property is successful. The false pipeline and data-transfer path detection property is successful. Finally, the completeness property is violated for the multiply operation. This operation is not defined in the *MUL5* unit. As a result, the multiply operation cannot execute in the pipeline. Once this is fixed, the validation of the DLX specification is successful.

Violation of Properties during DSE

We have performed many modifications of several architecture specifications. These are typical changes made by a designer during exploration. Here we briefly mention some of the errors captured using our approach.

- ✯ We have modified the MIPS R10K ADL description to include another load/store unit that supports only store operations. The false data-transfer path property is violated since there is a write connection from the load/store unit to the floating-point register file that will never be used.
- ✯ We have modified the PowerPC ADL description to have a separate L2 cache for instruction and data. Validation determined that there were no paths from L2 instruction cache to main memory. The connection between L2 instruction cache and unified L3 cache was missing.
- ✯ We have modified the C6x architecture's data memory by adding two SRAM modules to the existing cache hierarchy. The property validation fails due to the fact that the address ranges specified in the SRAMs and cache hierarchy are not disjoint, moreover union of these address ranges did not cover the physical address space specified by the processor description.
- ✯ We have added a coprocessor pipeline to the MIPS R10K architecture that supports vector integer multiplication. This path was reported as a false pipeline path since this opcode is not added in all the units in the path correctly. It also violates the completeness property since the read/write connections to integer register file is missing from the coprocessor pipeline.

- ☆ In the R10K architecture we have decided to use a coprocessor local memory instead of integer register file for reading operands. We have removed the read connections that are used to access the integer register file and added local memory, DMA controller and connections to the main memory. The connectedness property is violated for two ports in the integer register file.

- ☆ We have modified the PowerPC ADL description by reducing the instruction buffer size from 16 to 4. This violates the architecture-specific property. The fetch unit fetches 8 instructions per cycle and decode unit decodes 3 instructions per cycle, hence there is a potential for instruction loss.

Table 3.2 summarizes the errors captured during design space exploration of architectures. Each column represents one architecture and each row represents one property. An entry in the table presents the number of violations of that property for the corresponding architecture[1]. The figure (in parenthesis) below each architecture represents the number of design space explorations done for that architecture. Each class of problem is counted only once. For example, the DLX error mentioned above (where one of the unit has incorrect specification of the supported opcodes that led to false pipeline path for most of the opcodes) is counted only once instead of using the number of opcodes that violated the property.

Table 3.2: Summary of property violations during DSE

	ARM (1)	DLX (2)	C6x (2)	R10K (3)	PowerPC (2)
Connectedness	0	0	1	2	1
False Pipeline/Data-transfer Path	2	5	3	4	2
Completeness	1	2	3	3	2
Architecture-specific	2	4	5	12	6
Finiteness	0	0	0	1	1

Our experiments have demonstrated the utility of our validation approach across a wide range of realistic architectures, and the ability to detect errors in the architecture

[1]Of course the error numbers will change depending on the number of design space explorations and the type of modifications done each time.

specification, as well as errors generated through inconsistent modifications to an architecture during design space exploration.

3.2 Validation of Dynamic Behavior

This section presents a technique to verify the dynamic behavior of an architecture specified in an ADL by analyzing the instruction flow in the pipeline. Figure 3.7 shows our modified flow for validation of static and dynamic behaviors. The FSM model is generated from the ADL specification. Based on this model, we propose a method for validating pipelined processor specifications using two properties: determinism and in-order execution. Our automatic property checking framework determines if all the necessary properties are satisfied. In case of a failure, it generates traces so that a designer can modify the ADL specification of the architecture.

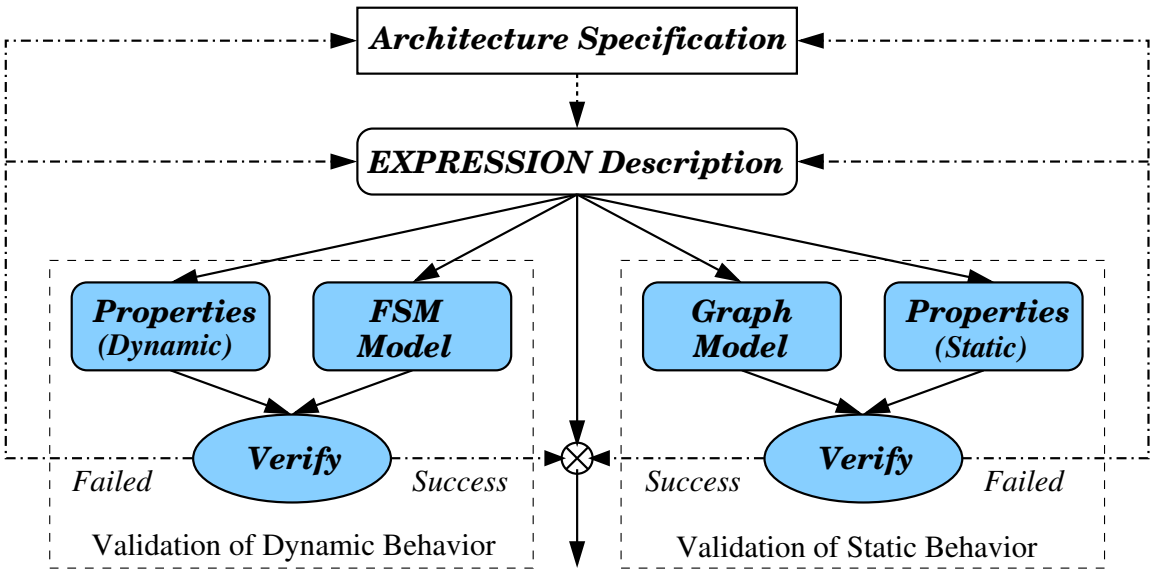


Figure 3.7: ADL driven validation of pipeline specifications

The remainder of this section is organized as follows. First, we describe a FSM-based modeling of processor pipelines. Next, we present the validation technique followed by a case study using the DLX architecture.

3.2.1 FSM-based Modeling of Processor Pipelines

In this section we describe how we derive the FSM model of the processor pipeline from the ADL specification. We first explain how we specify the information necessary for FSM modeling, then we present the FSM model of the processor pipelines using the information captured in the ADL.

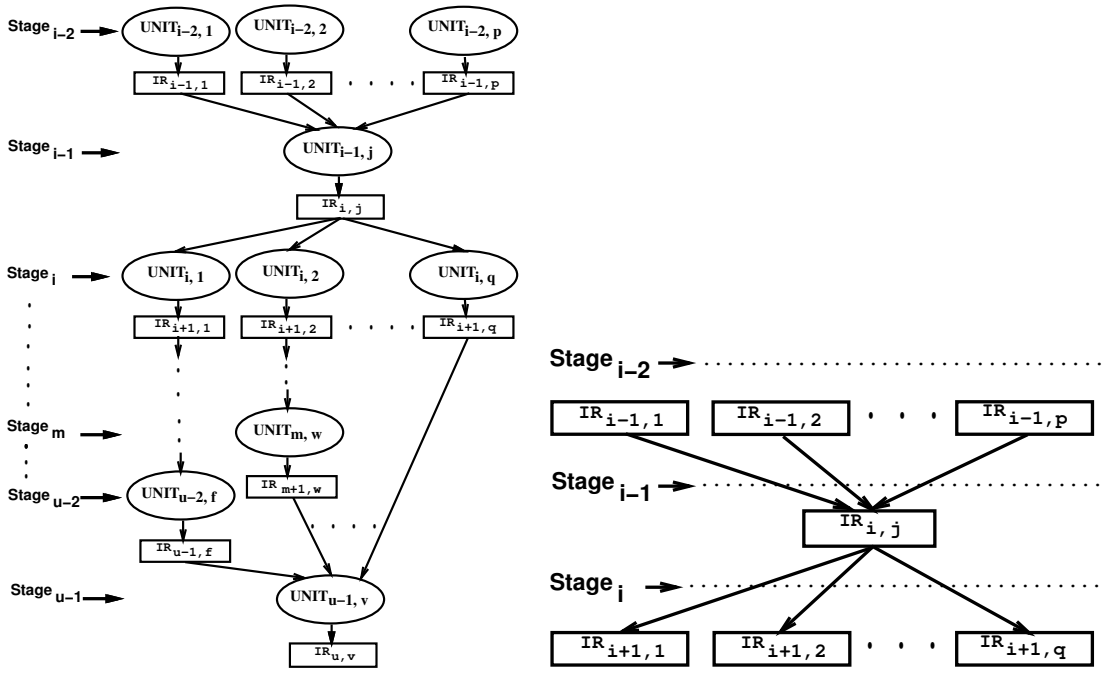
A. Processor Pipeline Description in ADL

Figure 3.8(a) shows a fragment of a processor pipeline. The oval boxes represent units, rectangular boxes represent pipeline latches, and arrows represent pipeline edges. In this section we briefly describe how we specify pipeline flow conditions for stalling, normal flow, bubble insertion, exception and squashing in the ADL.

A unit is in *normal flow* (NF) if it can receive instruction from its parent unit and can send to its child unit. A unit can be *stalled* (ST) due to external signals or due to conditions arising inside the processor pipeline. For example, the external signal that can stall a fetch unit is *ICache_Miss*; the internal conditions to stall the fetch unit can be due to decode stall, hazards, or exceptions. A unit performs *bubble insertion* (BI) when it does not receive any instruction from its parent (or busy computing in case of multicycle unit) and its child unit is not stalled. A unit can be in *exception* condition due to internal contribution or due to an exception. A unit is in bubble/nop *squashed* (SQ) stage when it has a nop instruction that gets removed or overwritten by an instruction of the parent unit.

For units with multiple children the flow conditions due to internal contribution may differ. For example, the unit $UNIT_{i-1,j}$ in Figure 3.8(a) with q children can be *stalled* when *any* one of its children is stalled, or when *some* of its children are stalled (designer identifies the specific ones), or when *all* of its children are stalled; or when *none* of its children are stalled. During specification, the designer selects from the set {*any, some, all, none*} the internal contribution along with any external signals to specify the stall condition for each unit. Similarly, the designer specifies the internal contribution for other flow conditions [68].

The PC unit can be *stalled* (ST) due to external signals such as cache miss or when



(a) With units and pipeline latches (b) With only instruction registers

Figure 3.8: A fragment of the processor pipeline

the fetch unit is stalled. When a branch is taken the PC unit is said to be in *branch taken* (BT) state. The PC unit is in *sequential execution* (SE) mode when the fetch unit is in normal flow, there are no external interrupts, and the current instruction is not a branch instruction.

B. FSM Model of Processor Pipelines

This section presents an FSM-based modeling of controllers in pipelined processors. Intuitively, the FSM captures the information of all the storage elements in the pipeline including program counter and pipeline latches. Let us assume that there are n such elements. Therefore, a state S_t^n in the FSM has the values of all the n elements at time t. The state transition (next-state) function returns the set of values of all the n elements at next time step (clock cycle). In other words the next state of S_t^n is S_{t+1}^n. The remainder of this section describes the FSM model in detail.

Figure 3.8(b) shows a fragment of the processor pipeline with only instruction registers[2] (IR). We assume a pipelined processor with in-order execution as the target for modeling and validation. The pipeline consists of n stages. Each stage can have more than one pipeline register (in case of fragmented pipelines). Each single-cycle pipeline register takes one cycle if there are no pipeline hazards. A multi-cycle pipeline register takes m cycles during normal execution (no hazards). Let $Stage_i$ denote the i-th stage where $0 \leq i \leq n-1$, and n_i the number of pipeline registers between $Stage_{i-1}$ and $Stage_i$. Let $IR_{i,j}$ denote an instruction register between $Stage_{i-1}$ and $Stage_i$ ($1 \leq i \leq n$, $1 \leq j \leq n_i$). The first stage, i.e., $Stage_0$, fetches an instruction from instruction memory pointed by program counter PC, and stores the instruction into the first instruction register $IR_{1,j}$ ($1 \leq j \leq n_1$). Without loss of generality, let us assume that $IR_{i,j}$ has p parent units and q children units as shown in Figure 3.8(b). During execution, the instruction stored in $IR_{i,j}$ is executed at $Stage_i$ and then stored into the next instruction register $IR_{i+1,k}$ ($1 \leq k \leq q$).

In this section, we define a state of the n-stage pipeline as values of PC and $\sum_{i=1}^{n-1} n_i$ instruction registers. Let $PC(t)$ and $IR_{i,j}(t)$ denote the values of PC and $IR_{i,j}$ at time t, respectively. Then, the state of the pipeline at time t is defined as

$$S(t) \;=\; < PC(t), IR_{1,1}(t), \cdots, IR_{i,j}(t), \cdots, IR_{n-1,n_{n-1}}(t) > \tag{3.5}$$

We first describe the flow conditions for stalling(ST), normal flow(NF), bubble insertion(BI), bubble squashing (SQ), sequential execution(SE), and branch taken (BT) in the FSM model, then we describe the state transition functions possible in the FSM model using the flow conditions.

In this section we use the symbol '$\vee$' to denote logical *or*, and '$\wedge$' to denote logical *and*. For example, $(a \vee b)$ implies (a or b), and $(a \wedge b)$ implies (a and b). We use the symbols $\bigvee_i^j$ and $\bigwedge_i^j$ to denote sum and product of symbols respectively. For example, $\bigvee_{i=0}^{2} a_i$ implies $(a_0 \vee a_1 \vee a_2)$, and $\bigwedge_{i=0}^{2} a_i$ implies $(a_0 \wedge a_1 \wedge a_2)$.

[2]We refer to these pipeline latches (registers) as instruction registers since they are used to transfer instructions from one pipeline stage to the next.

Modeling Conditions in FSM

Let us assume that every instruction register $IR_{i,j}$ has an exception bit $XN_{IR_{i,j}}$, which is set when the exception condition ($cond^{XN}_{IR_{i,j}}$ say) is true. The $XN_{IR_{i,j}}$ has two components viz., exception condition when the children are in exception ($XN^{child}_{IR_{i,j}}$ say) and exception condition due to exceptions on $IR_{i,j}$ ($XN^{self}_{IR_{i,j}}$ say). More formally the exception condition at time t in the presence of a set of external signals $I(t)$ on $S(t)$ is, $cond^{XN}_{IR_{i,j}}(S(t), I(t))$ ($cond^{XN}_{IR_{i,j}}$ in short),

$$cond^{XN}_{IR_{i,j}} = XN_{IR_{i,j}} = XN^{child}_{IR_{i,j}} \vee XN^{self}_{IR_{i,j}} \tag{3.6}$$

For example, if the designer specified that *any* (see Section 3.2.1(A)) of the children are responsible for the exception on $IR_{i,j}$ i.e., $IR_{i,j}$ will be in exception condition if any of its children is in exception, the Equation (3.6) becomes:

$$XN_{IR_{i,j}} = (\bigvee_{k=1}^{q} XN_{IR_{i+1,k}}) \vee XN^{self}_{IR_{i,j}}$$

Similarly, the conditions for squashing (say $cond^{SQ}_{IR_{i,j}}$), stalling (say $cond^{ST}_{IR_{i,j}}$), normal flow (say $cond^{NF}_{IR_{i,j}}$) and bubble insertion (say $cond^{BI}_{IR_{i,j}}$) are shown below.

$$cond^{SQ}_{IR_{i,j}} = SQ_{IR_{i,j}} = NF^{parent}_{IR_{i,j}} \wedge ST^{child}_{IR_{i,j}} \wedge ((IR_{i,j}).opcode == nop) \tag{3.7}$$

$$cond^{ST}_{IR_{i,j}} = (ST^{child}_{IR_{i,j}} \vee ST^{self}_{IR_{i,j}}) \wedge \overline{XN_{IR_{i,j}}} \wedge \overline{SQ_{IR_{i,j}}} \tag{3.8}$$

$$cond^{NF}_{IR_{i,j}} = NF^{parent}_{IR_{i,j}} \wedge NF^{child}_{IR_{i,j}} \wedge \overline{ST^{self}_{IR_{i,j}}} \wedge \overline{XN_{IR_{i,j}}} \wedge \overline{SQ_{IR_{i,j}}} \tag{3.9}$$

$$cond^{BI}_{IR_{i,j}} = BI^{parent}_{IR_{i,j}} \wedge BI^{child}_{IR_{i,j}} \wedge \overline{ST^{self}_{IR_{i,j}}} \wedge \overline{XN_{IR_{i,j}}} \wedge \overline{SQ_{IR_{i,j}}} \tag{3.10}$$

Similarly the conditions for PC viz., $cond^{SE}_{PC}$ (SE: sequential execution), $cond^{BI}_{PC}$ (BI: bubble insertion), and $cond^{BT}_{PC}$ (BT: branch taken) can be described using the information available in the ADL. The $cond^{BT}_{PC}$ is true when a branch is taken or when an exception is taken. When a branch is taken, the PC is modified with the target address. When an exception is taken, the PC is updated with the corresponding interrupt service routine address. Let us assume that the BT_{PC} bit is set when the unit completes execution of a branch instruction and the branch is taken. Formally,

$$cond^{SE}_{PC}(S(t), I(t)) = NF^{child}_{PC} \wedge \overline{ST^{self}_{PC}} \wedge \overline{BT_{PC}} \wedge \overline{XN_{IR_{1,j}}} \tag{3.11}$$

$$cond^{ST}_{PC}(S(t), I(t)) = (ST^{child}_{PC} \vee ST^{self}_{PC}) \wedge \overline{BT_{PC}} \wedge \overline{XN_{IR_{1,j}}} \tag{3.12}$$

$$cond^{BT}_{PC}(S(t), I(t)) = (BT_{PC} \vee XN_{IR_{1,j}}) \tag{3.13}$$

Modeling State Transition Functions

In this section, we describe the next-state function of the FSM. Figure 3.8(b) shows a fragment of the processor pipeline with only instruction registers. If there are no pipeline hazards, instructions flow from IR (instruction register) to IR every m cycles (m = 1 for single-cycle IR). In this case, the instruction in $IR_{i-1,l}$ $(1 \leq l \leq p)$ at time t proceeds to $IR_{i,j}$ after m cycles (m is the *timing* of $IR_{i-1,l}$, and $IR_{i,j}$ has p parent latches and q child latches as shown in Figure 3.8(b)), i.e., $IR_{i,j}(t+1) = IR_{i-1,l}(t)$. In the presence of pipeline hazards, however, the instruction in $IR_{i,j}$ may be stalled, i.e., $IR_{i,j}(t+1) = IR_{i,j}(t)$. Note that, in general, any instruction in the pipeline cannot skip pipeline stages. For example, $IR_{i,j}(t+1)$ cannot be $IR_{i-2,v}(t)$ $(1 \leq v \leq n_{i-2})$ if there are no feed-forward paths.

The rest of this section formally describes the next-state function of the FSM. According to the Equation (3.5), a state of a $n-$stage pipeline is defined by $(M+1)$ registers (PC and M instruction registers where, $M = \sum_{i=1}^{n-1} n_i$). Therefore, the next state function of the pipeline can also be decomposed into $(M+1)$ sub-functions each of which is dedicated to a specific state register. Let f_{PC}^{NS} and $f_{IR_{i,j}}^{NS}$ $(1 \leq i \leq n-1,$ $1 \leq j \leq n_i)$ denote next-state functions for PC and $IR_{i,j}$ respectively. Note that in general $f_{IR_{i,j}}^{NS}$ is a function of not only $IR_{i,j}$ but also other state registers and external signals from outside of the controller. For the program counter, we define three types of state transitions as follows.

$$
\begin{aligned}
& PC(t+1) \\
&= f_{PC}^{NS}(S(t), I(t)) \\
&= \begin{cases} PC(t) + L & \text{if } cond_{PC}^{SE}(S(t), I(t)) = 1 \\ target & \text{if } cond_{PC}^{BT}(S(t), I(t)) = 1 \\ PC(t) & \text{if } cond_{PC}^{ST}(S(t), I(t)) = 1 \end{cases}
\end{aligned} \tag{3.14}
$$

Here, $I(t)$ represents a set of external signals at time t, L represents the instruction length, and $target$ represents the branch target address which is computed at a certain pipeline stage. The $cond_{PC}^{x}$'s $(x \in SE, BT, ST)$ are logic functions of $S(t)$ and $I(t)$ as described in Equation (3.11) - Equation (3.13), and return either 0 or 1. For example, if $cond_{PC}^{ST}(S(t), I(t))$ is 1, PC keeps its current value at the next cycle.

For instruction registers, $IR_{i,j}$ $(2 \leq i \leq n-1,\ 1 \leq j \leq n_i)$, we define five types of state transitions as follows. The state transitions for the first instruction register, $IR_{1,j}$, will have $IM(PC(t))$ in place of $IR_{i-1,l}(t)$, where $IM(PC(t))$ denotes the instruction pointed by the program counter (PC) in instruction memory (IM).

$$
\begin{aligned}
& IR_{i,j}(t+1) \\
&= f_{i,j}^{NS}(S(t), I(t)) \\
&= \begin{cases}
IR_{i-1,l}(t) & \text{if } cond_{IR_{i,j}}^{NF}(S(t), I(t)) = 1 \\
IR_{i,j}(t) & \text{if } cond_{IR_{i,j}}^{ST}(S(t), I(t)) = 1 \\
nop & \text{if } cond_{IR_{i,j}}^{BI}(S(t), I(t)) = 1 \\
IR_{i-1,l}(t) & \text{if } cond_{IR_{i,j}}^{SQ}(S(t), I(t)) = 1 \\
nop & \text{if } cond_{IR_{i,j}}^{XN}(S(t), I(t)) = 1
\end{cases}
\end{aligned}
\tag{3.15}
$$

The $IR_{i,j}$ is said to be stalled at time t if $cond_{IR_{i,j}}^{ST}(S(t), I(t))$ is 1, resulting in $IR_{i,j}(t+1) = IR_{i,j}(t)$. Similarly, $IR_{i,j}$ is said to flow normally at time t if $cond_{IR_{i,j}}^{NF}(S(t), I(t))$ is 1. A *nop* instruction (bubble) is inserted in $IR_{i,j}$ when $cond_{IR_{i,j}}^{BI}(S(t), I(t))$ or $cond_{IR_{i,j}}^{XN}(S(t), I(t))$ is 1, resulting in $IR_{i,j}(t+1) = nop$. Similarly, when $cond_{IR_{i,j}}^{SQ}$ is 1, the bubble in $IR_{i,j}$ gets overwritten by the instruction from the parent instruction register, i.e., $IR_{i,j}(t+1) = IR_{i-1,l}(t)$ $(1 \leq l \leq n_{i-1})$.

In this FSM model, signals coming from the datapath or the memory subsystem into the pipeline controller are modeled as primary inputs to the FSM, and control signals to the datapath or the memory subsystem are modeled as outputs from the FSM.

3.2.2 Validation of Dynamic Properties

Based on the FSM model presented in Section 3.2.1, we propose a method for validating dynamic behaviors of pipelined processor specifications using two properties: determinism and in-order execution. We consider validation of dynamic behavior for architectures with in-order execution. We first describe the properties needed for validating the specification. Next, we present an automatic property checking framework driven by the EXPRESSION ADL [20].

A. Properties

This section presents two properties: determinism and in-order execution. Any pipelined processor with in-order execution must satisfy these properties.

Determinism

To ensure correct execution, there should not be any instruction or data loss in the pipeline. The bubble squashing and flushing of instructions are permitted. The flushed instructions are fetched and executed again. The next-state functions for all state registers must be deterministic. This property is valid if all the following equations hold for $\forall i, j (1 \leq i \leq n-1, 1 \leq j \leq n_i)$.

$$cond_{PC}^{SE} \vee cond_{PC}^{BT} \vee cond_{PC}^{ST} = 1 \tag{3.16}$$

$$cond_{IR_{i,j}}^{NF} \vee cond_{IR_{i,j}}^{ST} \vee cond_{IR_{i,j}}^{BI} \vee cond_{IR_{i,j}}^{XN} \vee cond_{IR_{i,j}}^{SQ} = 1 \tag{3.17}$$

$$\forall x, y (x, y \in \{SE, BT, ST\} \ \wedge \ x \neq y), \ cond_{PC}^{x} \wedge cond_{PC}^{y} = 0 \tag{3.18}$$

$$\forall x, y (x, y \in \{NF, ST, BI, XN, SQ\} \ \wedge \ x \neq y), \ cond_{IR_{i,j}}^{x} \wedge cond_{IR_{i,j}}^{y} = 0 \tag{3.19}$$

The first two equations mean that in the next-state function for each state register, the five conditions must cover all possible combinations of processor states $S(t)$ and external signals $I(t)$. The last two guarantee that any two conditions are disjoint for each next-state function. Informally, exactly one of the conditions should be true in a clock cycle for each state register. As a result, at any time t an instruction register will have a deterministic instruction. In other words, given an initial state of the pipelined processor and an input application program consisting of instruction sequences, it is possible to deterministically decide the instruction in a given instruction register at a given time t.

In-Order Execution

A pipelined processor with in-order execution is correct if all instructions that are fetched from instruction memory, flow from the first stage to the last stage, while maintaining their execution order. In order to guarantee in-order execution, state

transitions of adjacent instruction registers must depend on each other. Illegal combination of state transitions of adjacent stages are described below using Figure 3.8(b) where $2 \leq i \leq n-1$, $1 \leq j \leq n_i$, $1 \leq l \leq p$, and $1 \leq k \leq q$.

An instruction register can not be in normal flow if all the parent instruction registers (adjacent ones) are stalled. If such a combination of state transitions are allowed, the instruction stored in $IR_{i-1,l}$ at time t will be duplicated, and stored into both $IR_{i-1,l}$ and $IR_{i,j}$ in the next cycle. Therefore, the instruction will be executed more than once. Formally, the Equation (3.20) should be satisfied. Similarly, the equations (Equation (3.21) - Equation (3.32)) should be satisfied for $IR_{i,j}$. The detailed explanation is available in [68].

$$(\bigwedge_{l=1}^{p} cond^{ST}_{IR_{i-1,l}}) \wedge cond^{NF}_{IR_{i,j}} = 0 \tag{3.20}$$

$$cond^{NF}_{IR_{i,j}} \wedge (\bigwedge_{k=1}^{q} cond^{ST}_{IR_{i+1,k}}) = 0 \tag{3.21}$$

$$cond^{BI}_{IR_{i,j}} \wedge (\bigwedge_{k=1}^{q} cond^{ST}_{IR_{i+1,k}}) = 0 \tag{3.22}$$

$$cond^{NF}_{IR_{i-1,l}} \wedge cond^{BI}_{IR_{i,j}} = 0 \tag{3.23}$$

$$cond^{BI}_{IR_{i-1,l}} \wedge cond^{BI}_{IR_{i,j}} = 0 \tag{3.24}$$

$$cond^{ST}_{IR_{i-1,l}} \wedge cond^{SQ}_{IR_{i,j}} = 0 \tag{3.25}$$

$$cond^{XN}_{IR_{i-1,l}} \wedge cond^{SQ}_{IR_{i,j}} = 0 \tag{3.26}$$

$$cond^{SQ}_{IR_{i,j}} \wedge cond^{NF}_{IR_{i+1,k}} = 0 \tag{3.27}$$

$$cond^{SQ}_{IR_{i,j}} \wedge cond^{NI}_{IR_{i+1,k}} = 0 \tag{3.28}$$

$$cond^{NF}_{IR_{i-1,l}} \wedge cond^{XN}_{IR_{i,j}} = 0 \tag{3.29}$$

$$cond^{ST}_{IR_{i-1,l}} \wedge cond^{XN}_{IR_{i,j}} = 0 \tag{3.30}$$

$$cond^{SQ}_{IR_{i-1,l}} \wedge cond^{XN}_{IR_{i,j}} = 0 \tag{3.31}$$

$$cond^{BI}_{IR_{i-1,l}} \wedge cond^{XN}_{IR_{i,j}} = 0 \tag{3.32}$$

The above equations are not sufficient to ensure in-order execution in fragmented pipelines. An instruction I_a should not reach join node earlier than an instruction I_b when I_a is issued by the corresponding fork node later than I_b. Formally the following

equation should hold:

$$\forall(F, J), I_a \preceq_J I_b \Rightarrow \Gamma_F(I_a) < \Gamma_F(I_b) \tag{3.33}$$

where, $(F,\ J)$ is fork-join pair, $I_a \preceq_J I_b$ implies I_a reached join node J before I_b, $\Gamma_F(I_a)$ and $\Gamma_F(I_b)$ returns the timestamps when instructions I_a and I_b (respectively) are issued by the fork node F.

The previous property ensures that instruction does not execute out-of-order. However, with the current modeling two instructions with different timestamp can reach the join node. If join node does not have capacity for more than one instruction this may cause instruction loss. We need the following property to ensure that only one immediate parent of the join node is in normal flow at time t:

$$\forall x, y(x, y \in \{1, 2, ..., p\} \ \wedge \ x \neq y),\ cond^{NF}_{IR_{i-1,x}} \wedge cond^{NF}_{IR_{i-1,y}} = 0 \tag{3.34}$$

Similarly, the state transition of PC must depend on the state transition of $IR_{1,j}$ $(1 \leq j \leq n_1)$. The illegal combination of state transitions between PC and $IR_{1,j}$ are described below.

$$cond^{ST}_{PC} \wedge cond^{NF}_{IR_{1,j}} = 0 \tag{3.35}$$

$$cond^{SE}_{PC} \wedge (\bigwedge_{j=1}^{n_1} cond^{ST}_{IR_{1,j}}) = 0 \tag{3.36}$$

$$cond^{BT}_{PC} \wedge (\bigwedge_{j=1}^{n_1} cond^{ST}_{IR_{1,j}}) = 0 \tag{3.37}$$

$$cond^{SE}_{PC} \wedge cond^{BI}_{IR_{1,j}} = 0 \tag{3.38}$$

$$cond^{BT}_{PC} \wedge cond^{BI}_{IR_{1,j}} = 0 \tag{3.39}$$

$$cond^{SE}_{PC} \wedge cond^{XN}_{IR_{1,j}} = 0 \tag{3.40}$$

$$cond^{ST}_{PC} \wedge cond^{SQ}_{IR_{1,j}} = 0 \tag{3.41}$$

$$cond^{ST}_{PC} \wedge cond^{XN}_{IR_{1,j}} = 0 \tag{3.42}$$

We have described all possible illegal combination of state transition functions (Equation (3.20) - Equation (3.42)). However, Equation (3.23), Equation (3.24), Equation (3.27), and Equation (3.28) are not necessary to prove in-order execution.

B. Automatic Validation Framework

Algorithm 6 describes the specification validation technique. It accepts the processor specification, described in EXPRESSION ADL [20], as input. The FSM model and the properties are generated from the ADL specification. In case of a failure, it generates counter-examples so that the designer can modify the ADL specification of the architecture.

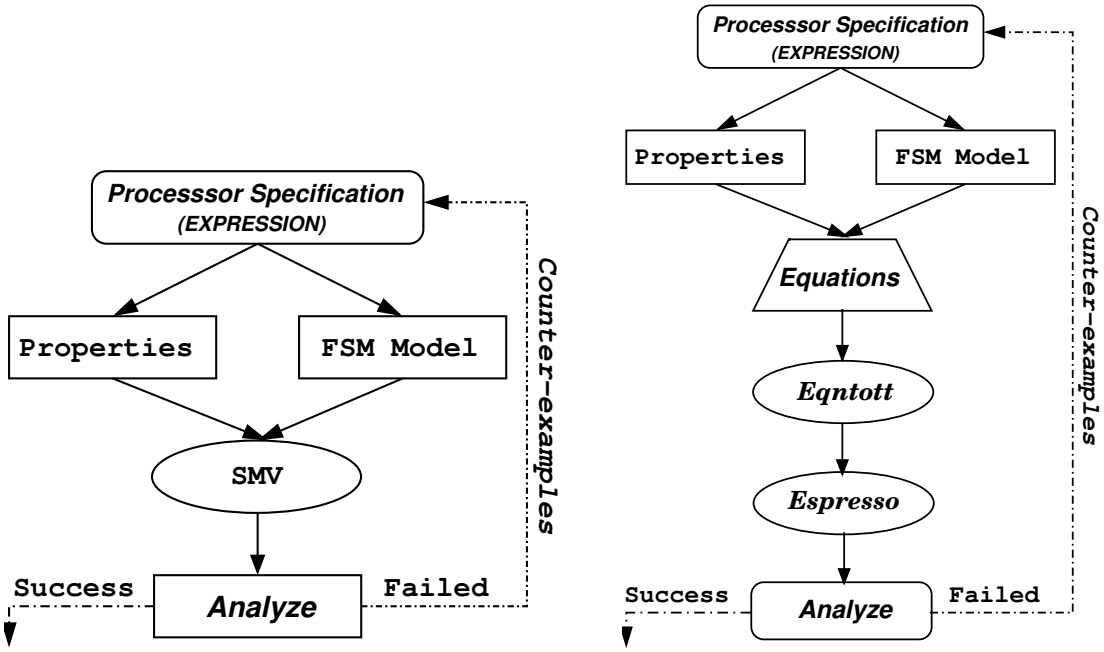
Algorithm 6: *Validation of Pipeline Specification*
Input: ADL specification of the processor architecture.
Outputs: *Success*, if the processor model satisfies the properties.
Failure otherwise, and produces the counter-examples.
Begin
Generate FSM model from the ADL specification using Equation (3.5) - Equation (3.15)
Generate properties using Equation (3.16) - Equation (3.42)
Apply the properties on the FSM model to verify determinism and in-order execution.
Return *Success* if all the properties are verified;
Failure otherwise, and produce the counter-example(s).
End

We have verified the properties using two different approaches. First, we have used an SMV [28] based property checking framework as shown in Figure 3.9(a). The SMV based approach fits nicely in our validation framework. However, the SMV is limited by the size of the design it can handle. We have also developed an equation solver based framework as shown in Figure 3.9(b) that can handle complex designs. In this section, we briefly describe these two approaches. The detailed description is available in [68].

Validation using Model Checker

The FSM model (SMV description) of the processor is generated from the ADL specification. The properties are also described using SMV description. The properties are applied on the FSM model using the SMV model checker as shown in Figure 3.9(a). In case of failure, SMV generates counter-examples that can be used to

modify the ADL specification. Each counter-example describes the failed equation(s) and the instruction registers that are involved.



(a) Framework using SMV

(b) Using equation solver

Figure 3.9: Automatic validation frameworks

We have verified the in-order execution style of the processor specification in two ways. First, the framework generates properties using Equation (3.20) - Equation (3.42) to verify in-order execution. This is similar to how other properties (e.g., determinism) are verified. Second, an auxiliary automata is used instead of using equations to verify in-order execution. In the auxiliary automata based approach, we use the same FSM model of the processor (SMV description) generated from the ADL specification. We have developed a SMV module that generates two instructions randomly with random delay between them. These two instructions are recorded and fed to the FSM model. The processor (FSM) model accepts these instructions and performs regular computations. At the completion (e.g., writeback unit) the auxiliary automata analyzes these two instructions to see whether they completed in the same sequence as generated. Note that, this auxiliary automata does not need any manual

modification for different architectures. In case of failure, SMV generates counter-examples containing instruction sequences (instruction pair with NOPs in between them) that violate in-order execution for the processor model.

Validation using Equation Solver

In the second approach, the framework generates the FSM model and flow equations for normal flow, stall, exception, squashing, and bubble insertion for each instruction register and sequential execution, stall, and branch taken for PC using ADL specification and Equation (3.5) - Equation (3.15). It generates the equations necessary for verifying properties using ADL description and Equation (3.16) - Equation (3.42) as shown in Figure 3.9(b).

The *Eqntott* [26] tool converts these equations in two-level representation of a two-valued Boolean function. This two-level representation is fed to *Espresso* [25] tool that produces minimal equivalent representation. Finally, the minimized representation is analyzed to determine whether the property is successfully verified or not. In case of failure, it generates traces explaining the cause of failure. The trace contains the equation(s) that failed, and the identification of the instruction registers involved. The designer therefore knows the property that is violated and the reason for the violation. This information is used to modify the ADL specification. The detailed description is available in [68].

3.2.3 A Case Study

In a case study we successfully applied the proposed methodology to the single-issue DLX [22] processor. We have chosen the DLX processor since it has been well studied in academia and contains many interesting features such as fragmented pipelines and multicycle units.

We used the EXPRESSION ADL [20] to capture the structure and behavior of the DLX processor shown in Figure 3.6. We captured the conditions for stalling, normal flow, exception, branch taken, squashing, and bubble insertion in the ADL. Using the ADL description, we automatically generated the equations for flow conditions for all

the units. The necessary equations for verifying the properties such as determinism and in-order execution are generated automatically from the given ADL specification. The detailed description of the case study is available in [68].

We have used *Espresso* [25] to minimize the equations. These minimized equations are analyzed to verify whether the properties are violated. Our framework determined that the Equation (3.33) is violated and generated a simple instruction sequence which violates in-order execution: floating-point addition followed by integer addition. The decode unit issued the floating point addition I_{fadd} operation in cycle m to floating-point adder pipeline and an integer addition operation I_{iadd} to integer ALU at cycle *m+1*. The instruction I_{iadd} reached the join node (MEM unit) prior to I_{fadd}.

We have modified the ADL specification to change the stall condition depending on current instruction in decode unit and the instructions active in the integer ALU, MUL, FADD, and DIV pipelines. The current instruction will not be issued (decode stalls) if it leads to out-of-order execution. Our framework generates equations for the modified processor model. The Equation (3.34) is violated for this modeling for the join node (MEM unit). The instruction sequence generated by our framework for this failure consists of a multiplication operation (issued by decode unit in cycle m) followed by a floating-point add operation (issued by decode unit in cycle *(m + 3)*). As a result both the operations reached memory stage at cycle *(m+7)*.

Finally, the stall condition of the decode unit is modified to avoid completion of two instructions at the same time. The in-order execution was successful for this modeling. In such a simple situation this kind of specification mistakes might appear as trivial, but when the architecture gets complicated and exploration iterations and varieties increase, the potential for introducing bugs also increases.

We have verified the properties using two different methods: using the *SMV* model checker and the *Espresso* equation solver, as described in Section 3.2.2. We have used a 300 MHz Sun UltraSparc-II with 1024M RAM to run the experiments. Table 3.3 shows the performance of the two methods for verifying in-order execution property. We have used the VLIW DLX architecture as the base configuration and modified the number of opcodes. The first column presents our two methods of specification validation. The second, third, and fourth columns present the execution time (in

seconds) of the two methods for verifying in-order execution property for different architecture configurations.

Table 3.3: Validation of in-order execution by two frameworks

	DLX Processor Configurations		
	8 opcodes	16 opcodes	32 opcodes
SMV based Framework	302.4 sec	400.4 sec	740.9 sec
Espresso based Framework	5.4 sec	6.7 sec	9.4 sec

We have performed experiments by modifying the pipeline structure such as addition of pipeline paths and pipeline stages. Our SMV based framework could not verify in-order execution when pipeline path is added to the VLIW DLX architecture. However, our equation solver based framework requires in the order of seconds to verify in-order execution and can handle complex configurations. The SMV based framework performed better for verifying the determinism. This is due to the fact that the properties (equations) that need to be applied to verify determinism consists of local computations for each state register. The SMV based framework took 0.8 seconds to verify determinism property, whereas the equation solver based framework took 4 seconds for the same DLX configuration. Although we have not applied this technique on other architectures, we believe the SMV based framework is suitable for verifying determinism property whereas our equation solver based framework can be used for verifying in-order execution of complex architectures.

3.3 Chapter Summary

Validation of the architectural specification is essential to ensure that the reference model is golden so that it can be used to uncover bugs in the design. This chapter presented a framework for automatic modeling and validation of pipelined processor specifications driven by an architecture description language (ADL).

We developed validation techniques to ensure that the static behavior of the pipeline is well-formed by analyzing the structural aspects of the specification using a graph based model. We applied these techniques on the graph model of the

MIPS R10K, TI C6x, DLX, and PowerPC architectures to demonstrate the usefulness of this approach. The dynamic behavior is verified by analyzing the instruction flow in the pipeline using a FSM-based model to validate several important architectural properties such as determinism and in-order execution in the presence of hazards and multiple exceptions. We applied this methodology to the DLX processor to demonstrate the usefulness of this technique.

These properties are by no means complete to prove the correctness of the specification. The designer can add new architecture specific properties and easily integrate it in our framework. Our validation framework uses two approaches: SMV based property checking and Espresso based equation minimization. The framework determines whether all the necessary properties are satisfied. In case of a failure, it generates traces so that a designer can modify the ADL specification of the architecture.

Chapter 4

Model Generation using Functional Abstraction

Contemporary processor architectures vary widely in terms of their architectural features. Program address generation and instruction dispatch features are widely used in DSP processors. VLIW processors use strong compiler support to ensure correct execution of long instruction words. Superscalar processors on the other hand, use hardware scheduling techniques, register renaming, and so on. Multimedia processors support SIMD operations. Furthermore, each architecture has different branch prediction schemes, execution style (e.g., in-order, out-of-order), interrupt handling procedures, and last but not the least different memory subsystems. Emerging architectures have combined features of classical architectures (e.g., RISC, DSP, VLIW, and superscalar). For example, the Intel Itanium combines the features of VLIW and superscalar architectures; the TI C6x family combines the features of DSP and VLIW architectures. In order to allow rapid design space exploration of such heterogeneous processor-memory architectures, the framework requires the ability to capture a wide variety of such architectural features.

Moreover, during design space exploration using customized Intellectual Property (IP) cores designers may want to add certain architectural features (e.g., some superscalar features to a VLIW processor core) to see how it impacts the area, power, performance, and other important design parameters. Similarly, to find the best

match between the application characteristics and the memory organization features (e.g., caches, stream buffers, access modes, SRAM, DRAM etc.), the designer needs to explore different memory configurations in combination with different processor architectures, and evaluate each such system for cost, power, and performance. To enable this, designers need (i) a way of specifying a wide variety of processor-memory features and (ii) the ability to generate automatically executable models of the architecture. It is necessary to find the commonality (if exists) among such heterogeneous architectures and use that as a building block for defining a set of abstraction primitives. The abstraction primitives should be simple enough to allow correlation with the architectural features. On the other hand, the primitives should be generic enough to be useful across a wide range of architectures. In this chapter we present a functional abstraction technique that enables automatic generation of executable models from the ADL specification.

This chapter is organized as follows. Section 4.1 surveys contemporary processor-memory architectures. Section 4.2 presents the functional abstraction needed to capture a wide variety of architectural features and memory configurations. Section 4.3 describes the procedure for reference model generation from the ADL specification using functional abstraction. Finally, Section 4.5 summarizes the chapter.

4.1 Survey of Contemporary Architectures

We have studied contemporary processor and memory architectures from popular architectural domains, including RISC, DSP, VLIW, and superscalar [49]. This section summarizes the survey and outlines the similarities and differences of the architectural features available in a wide a variety of processor and memory architectures.

4.1.1 Summary of Architectures Studied

In order to understand and characterize the diversity of contemporary architectures, we have surveyed processors from different architectural domains - RISC (MIPS

R4000 [48] and StrongArm [82]), DSP (Motorola 56000 and TI C5x), VLIW DSP (TI C6x [33], MAP1000A [7], and Motorola StarCore [29]), superscalar (MIPS R10000 [31], MPC7450 [30], Sun UltraSparc IIi [83], and DEC Alpha 21364), and hybrid (Intel IA-64 [95]). The Intel IA-64 architecture has combined features of VLIW and superscalar processors with out-of-order execution.

Table 4.1 summarizes the processor-memory features for different architectures. Each row of the table corresponds to an architectural feature. Each column represents an architecture. We have shown the relationship between a feature and an architecture only. In general, an architectural feature may also depend on the type of the instruction.

An entry in Table 4.1, *TAB[F, A]*, represents the behavior of an architecture A towards a feature F. If an entry is marked x, that feature is supported by that architecture. If an entry is blank, the feature is either not supported or not applicable (or not known) to that architecture. An entry containing an integer number, n, means that feature is supported n times. An entry containing a series , *(n-m)*, implies that the feature is supported for i times, where $(n <= i <= m)$. Similarly, an entry containing a set, $\{n,m\}$, means that the feature is supported either n or m times. For example, the table entry with memory feature "*Levels of D-Cache*" and processor name *IA64* has value 3, this implies IA-64 has 3 levels of data cache. The row corresponding to "*operand read in*" has four types of values depending on where the operands are read in the pipeline: (D: Decode stage), (R: Read stage), (I: Issue stage), and (E: Execute stage). The row corresponding to *Branch Prediction* has values that indicate the method of branch prediction employed in the respective architecture: (2b: 2-bit algorithm using branch history table), (BT: BTB based prediction), and (MA: dynamically choose among multiple algorithms based on local predictor table, global predictor table and branch history table).

4.1.2 Similarities and Differences

Broadly speaking, the structure of a processor consists of functional units, connected using ports, connections and pipeline latches. Major functional units are the

Architectures	**RISC**		**DSP**		**VLIW DSP**			**Superscalar**				**Hybrid**
Processor-Memory Features	R4K	SA	56K	C5x	C6x	MA	SC	R10	MP	U3	α64	IA64
# of fetches/cycle	2	1	1	1	8	4	8	4	4	4	4	6
# of fetch stages	2	1	1	1	4		3	1	2	1	1	2
# of decodes/cycle	2	1	1	1	8	4		4	3	4	4	
# entries in decode RS									12			8
# of issue units								3	3	1	3	3
# of issues/cycle							6	5	6	4	6	6
# entries in issue RS								48	12		35	
# operations/instruction	1	1	1	1	8	4		1	1	1	1	
# of parallel exec units					8	4	6	5	11		6	
Branch Prediction								2b	BT		MA	
Feedback paths		x								x		x
Operand read in	D	D	E	R	E	E	E	I	I	I	R	I
SIMD support						x			x	x		x
# entries in completion Q								32	16			
Register Renaming								x	x		x	x
Dynamic Scheduling								x	x	x	x	x
Speculation											x	x
Predication						x						x
# register files	2	1	3	1	3	3	2	2	3		3	5
# Coprocessors	3	1										
# pipeline stages	8	5	3	4	3		5	5-7	7	9	6	10
Levels of D-Cache	1-2	1			0-2	1	0-2	2	3	2	2	3
cache prefetch		x								x		x
cache hints												x
On-chip SRAM		x			x	x	x			x		
configurable SRAM					x							
Off-chip DRAM	x	x	x	x	x	x	x	x	x	x	x	x
page/burst mode		x			x							
Write Buffer		x								x	x	
Read Buffer		x										
Victim Buffer											x	
Stack			x	x			x					
FIFO						x						
DMA		x			x	x	x				x	
parallel mem transfers	1		2		2		2	1			2	2
mem pipelining								x			x	

Table 4.1: Processor-memory features of different architectures. *R4K: MIPS R4000, SA: StrongArm, 56K: Motorola 56K, c5x: TI C5x, c6x: TI C6x, MA: MAP1000A, SC: Starcore, R10: MIPS R10000, MP: Motorola MPC7450, U3: SUN UltraSparc IIi, α64: Alpha 21364, IA64: Intel IA-64*

PC unit, fetch unit, decode unit, branch prediction unit, issue unit, load store unit, TLB, execute unit and completion or writeback unit. Similarly, the structure of a memory subsystem consists of SRAM, DRAM, cache hierarchy, and so on.

Although a broad classification makes the architectures look similar, each architecture differs in terms of the algorithm it employs in branch prediction, the way it detects hazards, the way it handles exceptions etc. Moreover, each unit has different parameters for different architectures (e.g., number of fetches per cycle, levels of cache, cache line size etc.). Program address generation and instruction dispatch features are widely used in DSP processors. VLIW processors use strong compiler support to ensure correct execution of long instruction words. Superscalar processors on the other hand, use hardware scheduling techniques, register renaming etc. Multimedia processors support SIMD operations. The contemporary EPIC architectures use predication and speculation techniques to increase instruction level parallelism.

Depending on the architecture a functional unit may perform the same computation at different stages in the pipeline. For example, read-after-write (RAW) followed by operand read happen in the decode unit for some architectures (e.g., DLX [22]), whereas in some others these operations are performed in the issue unit (e.g., MIPS R10K [31]). Some architectures even allow operand read in the execution unit. On the other hand, some architectures do not issue operations if RAW hazard is detected while others issue the operation in spite of RAW hazard (e.g., use snooping to read the data at execution stage using feedback paths). In other words, the same functionality is used at different stages in the pipeline for different architectures.

Towards obtaining a unifying abstraction, we can observe some fundamental differences from the study; the architecture may use:

1. the same functional or memory unit with different parameters
2. the same functionality in different functional or memory units
3. 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 that can be used by different architectures at different stages in the pipeline. The last one is difficult to alleviate since it is new, unless this new functionality can be composed of existing sub-functions. Section 4.2 presents the functional abstraction needed to capture a wide variety of architectural features and memory configurations.

4.2 Functional Abstraction

Functional abstraction allows the system designer to describe a wide variety of architectures. In this section we present functional abstraction by way of illustrative examples. We first explain the functional abstraction needed to capture the structure and behavior of the processor and memory subsystem, then we discuss the issues related to defining a generic controller functionality, and finally we discuss the issues related to handling interrupts and exceptions.

4.2.1 Structure of a Generic Processor

The structure of each functional unit is captured using parameterized functions. However, generic functions are not sufficient since each functional unit may perform a different set of computations depending on the architecture. Hence, there is a need for parametric sub-functions. Based on the observations made in Section 4.1, we have defined a set of common functions and sub-functions with appropriate parameters. First, we describe the generic functions. Next, we describe the generic sub-functions. Finally, we discuss how these functions and sub-functions are used to compose a new processor architecture.

Generic functions

We capture the structure of each functional unit using parameterized functions. For example, a fetch unit functionality contains several parameters, such as number of operations read per cycle, number of operations written per cycle, reservation station size, branch prediction scheme, number of read ports, number of write ports, and so on. Figure 4.1 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 fetch unit reads m operations from the reservation station and writes them to the output latch (fetch to decode latch) and uses BTB based branch prediction mechanism.

```
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);
}
```

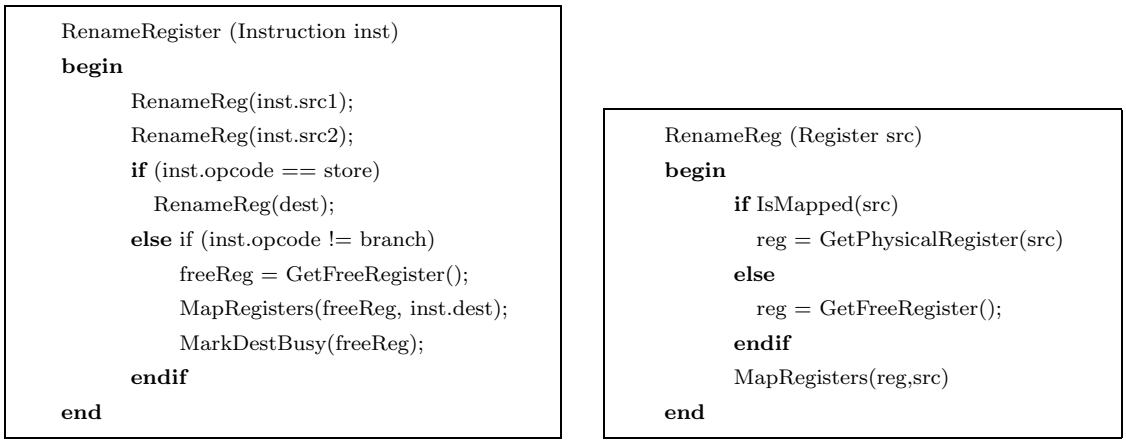
Figure 4.1: A fetch unit example

We have defined parameterized functions for all functional units present in contemporary programmable architectures including fetch unit, branch prediction unit, decode unit, issue unit, execute unit, completion unit, interrupt handler unit, PC Unit, Latch, Port, Connection, and so on. The detailed description of each of the generic functions is available in [49].

Generic sub-functions

We have defined sub-functions for all the common activities e.g., ReadLatch, WriteLatch, ReadOperand, and so on. Table 4.2 lists some of the common activities that we have identified. The first column represents the name of the function, the second column describes the activity, and the last column describes the input and output parameters of the function. We have also defined a set of sub-functions

including *RenameRegister* and *GraduateOperation* using sub-functions. Figure 4.2 shows a specific implementation of *RenameRegister* modeled using sub-functions.



(a) Rename registers in an instruction

(b) Rename a register

Figure 4.2: Modeling of RenameRegister function using sub-functions

New architecture generation using generic functions and sub-functions

So far we have discussed the generic functions and sub-functions necessary to capture a wide variety of processor architectures. In this section we briefly describe how these functions and sub-functions can be used to compose a new architecture or modify an existing architecture. First, we describe how to compose a simple RISC architecture. Next, we discuss what generic functions are necessary to modify the simple RISC architecture into a VLIW or superscalar architecture.

A RISC architecture typically has four pipeline stages: fetch, decode, execute, and writeback. Each of these stages requires one generic function. Each function uses *ReadLatch* sub-function to read the instruction from the pipeline latch. At the end of the computation each function uses *WriteLatch* sub-function to write the modified instruction into the output latch. The fetch function reads the program counter (PC) value using *ReadPC*. If the architecture supports branch prediction, the fetch function needs to use appropriate sub-functions such as *QueryPredictor*, *QueryBTB*, and so on. Depending on the outcome, the fetch function either invokes

Table 4.2: A list of common sub-functions

Function Name	**Description**	**Parameters**
ReadLatch	Read a latch for n operations	Latch X, n, Data
WriteLatch	Write data to a latch	Latch, Data
QueryPredictor	Query prediction status	Branch address, status
QueryBTB	Query predicted address	Branch and memory address
UpdateBTB	Send address to branch unit	ID, target address
UpdatePredictor	Update branch predictor	ID, prediction type
BranchOther	Other branch address	ID, Address
IncrementPC	Increase PC with X	X, New PC
SetPC	New PC address X	X, New PC
ReadPC	Get PC	PC address
RSInsertOperation	Add one operation to RS	Operation
RSInsertOperations	Add X operations to RS	Operations, X
RSDeleteOperation	Dequeue operation from RS	ID
RSReadOperation	Read one operation from RS	Operation
RSReadOperand	Read n's operations operands	RS, n, RS
ReadOperand	Read one operand	Address bus, Reg name, Data
WriteResult	Write operand	Data/Addr bus, Reg name, Data
MarkDestBusy	Mark Register busy	Register name
ReleaseDest	Unmark Register busy	Register name
CheckRAW	Check for RAW	Register name, status
CheckWAW	Check for WAW	ID, status
CheckWAR	Check for WAR	ID, status
IsUnitBusy	Is unit X busy	X, status
IsUnitStalled	Is unit X stalled	X, status
IsOperandRead	Is operand X read	ID, X, status
MarkOperandRead	Mark the operand as read	ID, X
HasUnitRS	Does unit X have RS?	X, status
SetUnitStalled	Set Stall bit for unit X	X, True/False
SetUnitBusy	Set Busy Bit for unit X	X, True/False
ReadPredicate	Check predicate register X	Pred reg. X, status
WritePredicate	Set predicate register X to Y	Pred reg., value
CheckPredicate	Query ID's predicate	ID, status
ExecuteOperation	Execute an operation	Src1, Src2, func, Result
ExecuteBranchOperation	Execute branch	Src1-2, func, Result, Cmp_reg
MarkOperationDone	Mark operation done in comp queue	ID
IsOperationDone	Query if operation done	ID, status
CompletionQDeleteOperation	Delete an entry from comp queue	ID
FlushCompletionQ	Remove all operations above ID	ID
IsOperationValid	Query if operation is valid	ID, status
SetValidBit	Set valid bit to X for operation	ID, X
IsBranchAhead	Is there a branch ahead?	ID, status
IsBranchOperation	Is operation a branch?	ID, status
IsStoreOperation	Is operation a store?	ID, status
IsMapped	Is X in mapping table	Reg X, status
GetPhysicalRegister	For a logical register	Logical, physical reg
GetFreeRegister	Return a free physical reg	Register number
MapRegisters	logical to physical	Logical, physical reg
ComputeBusybit	check if unit is busy	Incoming operations, free entries, cycles left

IncrementPC or *SetPC*. Similarly, the decode function uses *CheckRAW*, *CheckWAR*, and *CheckWAW* sub-functions to perform hazard detection. The source operands of the instruction are read using *ReadOperand* sub-function. The execute function uses *ExecuteOperation* sub-function to execute an operation. Finally, the writeback function uses *WriteResult* sub-function to write the result back into the register file.

To convert the RISC architecture into a VLIW one that can issue m operations per cycle to the n pipeline paths, we need to perform the following modifications to the functions discussed above. The decode function can use a reservation station (instruction buffer). The instruction buffer can be accessed using sub-functions such as *RSInsertOperation*, *RSDeleteOperation*, and so on. The decode function needs to use *IsUnitBusy* and *IsUnitStalled* sub-functions to detect structural hazard and decide where to send the next instruction. Each pipeline path needs to have separate execute functions. In case the architecture supports predicated execution, a set of sub-functions including *ReadPredicate*, *WritePredicate*, and *CheckPredicate* can be used.

To add superscalar features to the existing VLIW architecture, the following modifications can be done. The decode function can invoke *RenameRegister* to perform register renaming. Several sub-functions such as *GetPhysicalRegister* and *GetFreeRegister* are also useful in this regard. If the intended execution style is out-of-order execution, we need to add a completion queue (in-order buffer) in the architecture. The decode function needs to insert an operation in the queue using *CompletionQInsertOperation* before issuing it to the child unit. This is to ensure in-order completion in the presence of out-of-order execution. The writeback function can delete the front operation of the queue using *CompletionQDeleteOperation* sub-function. The completion queue can also be used to perform WAW and WAR checks, to flush necessary instructions in the pipeline, to enforce in-order completion of branches and memory writes, and to synchronize events such as all memory writes are completed and all pending exceptions are reported.

4.2.2 Behavior of a Generic Processor

The behavior of a generic processor is captured through the definition of operations. Each operation is defined as a function, with a generic set of parameters, that performs the intended functionality. The parameter list includes source and destination operands, and necessary control and data type information. We have defined common sub-functions (generic set) such as ADD, SUB, MUL, SHIFT, and so on [49].

```
ADD (src1, src2) {
  return (src1 + src2);
}
```

```
MUL (src1, src2) {
   return (src1 * src2);
}
```

```
MAC (src1, src2, src3) {
   return ( ADD( MUL(src1, src2), src3) );
}
```

Figure 4.3: Modeling of MAC operation

Given a new (target) operation and a mapping[1] between the target operation and the generic operations, the functionality of the new operation can be created using the functionalities of the existing operations. For example, the MAC (multiply and accumulate) functionality can be composed of two sub-functions (ADD and MUL) as shown in Figure 4.3.

4.2.3 Structure of a Generic Memory Subsystem

Each type of memory module, such as SRAM, cache, DRAM, SDRAM, stream buffer, and victim cache, is modeled using a function with appropriate parameters. For example, the cache function shown in Figure 4.4(a) has many parameters including cache size, line size, associativity, word size, replacement policy, write policy, and latency. It performs four operations: read, write, replace, and refill. These functions can have parameters for specifying pipelining, parallelism, access modes (normal read, page mode read, and burst read), and so on. Again, each function is composed of

[1]Such mappings are typically available at the specification level and used by a compiler to perform instruction selection.

sub-functions. For example, the associative cache function shown in Figure 4.4(b) is modeled using cache sub-function.

```
Cache(cache size, line size, ... opType, addr, data)
begin
    // It has three storages: tag, cache, valid
    // Get row, col, and tag from addr
    if opType is READ
      if ((tag[row] == tag) and valid[row])
        data = cache[row][col]
        return HIT
      else
        return MISS
      endif
    else if opType is WRITE
      .........
    else if opType is REPLACE
      .........
    else if opType is REFILL
      .........
    endif
end
```

(a) Cache function

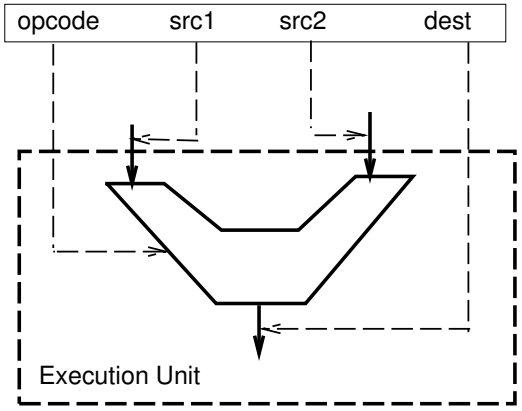
```
AssociativeCache (..., assoc, opType, addr, dataOut)
begin
    if opType is READ
      /** Find the one with data **/
      for ( ci=0; ci < associativity; ci ++)
          stat = Cache(cache_ci, ... READ, data)
          if stat is HIT
            dataOut = data
            return HIT
          endif
      endfor
      // Find the cache to be replaced and refilled
      Cache (..., cache, ..., REPLACE, addr)
      Cache (..., cache, ..., REFILL, addr)
      ......
    else if opType is WRITE
      ......
    endif
end
```

(b) Associative cache function

Figure 4.4: Modeling of associative cache function using sub-functions

4.2.4 Generic Controller

A major challenge in defining an architectural abstraction is the modeling of control for a wide range of architectural styles. We define control in both distributed and centralized manners. The distributed control is transfered through pipeline latches. While an instruction gets decoded the control information needed to select the operation, the source and the destination operands are placed in the output latch as shown in Figure 4.5(a). These decoded control signals pass through the latches between two pipeline stages unless they become redundant. For example, when the value for *src1* is read that particular control is not needed any more, instead the read value will be in the latch. We have shown here only the control information of the latch. The latch also contains data values and predicate registers (if applicable).



(a) Example of distributed control

Pipeline Stages	Pipeline Bandwidth (parallelism)			
		Fetch BB:0 SB:0		
		Decode BB:0 SB:0		
	IALU BB:0 SB:0	MUL1 BB:0 SB:0	FADD1 BB:0 SB:0	DIV BB:0 SB:0
		MUL2 BB:0 SB:0	FADD2 BB:0 SB:0	
		MUL3 BB:0 SB:0	FADD3 BB:0 SB:0	
		MUL4 BB:0 SB:0	FADD4 BB:0 SB:0	
		MUL5 BB:0 SB:0		
		MUL6 BB:0 SB:0		
		MUL7 BB:0 SB:0		
		MEM BB:0 SB:0		
		WriteBack BB:0 SB:0		

(b) Control Table for the DLX processor

Figure 4.5: Examples of distributed and centralized controllers

The centralized control is maintained by using a generic control table. The number of rows in the table is equal to the number of pipeline stages in the architecture. The number of columns is equal to the maximum number of parallel units present in any pipeline stage. Each entry in the control table corresponds to one particular unit in the architecture. It contains information specific to that unit e.g., busy bit (BB), stall bit (SB), list of children, list of parents, opcodes supported, and so on. For example, Figure 4.5(b) shows the control table for the DLX processor shown in Figure 3.6. The control table captures all the necessary details to perform selective or complete stalling of the pipelines. Stalling happens due to three kinds of hazards: structural hazards, data hazards, and control hazards.

4.2.5 Interrupts and Exceptions

Another major challenge in defining architectural abstractions is the modeling of interrupts and exceptions. We briefly describe the abstraction needed to capture a wide variety of exceptions and interrupts in programmable architectures. Each exception is captured using an appropriate sub-function. Opcode related exceptions

(e.g., divide by zero) are captured in the opcode functionality. Functional unit related exceptions (e.g., illegal slot exception) are captured in functional units. External interrupts (e.g., reset, debug exceptions) are captured in the control unit functionality.

We model an interrupt handler unit that services these exceptions. It has information regarding the priority of interrupts and which exceptions generate what interrupt. The generic interrupt handler has a parameterized priority table. The interrupt handler unit generates one particular interrupt based on the priority. Before execution of an interrupt service routine, context saving and complete/partial flushing occurs. The specific type of flushing is decided by the semantics of the interrupt: complete flushing clears the entire pipeline; partial flushing means flushing only the instructions behind the interrupted instruction and allowing the previous instructions to continue using the program order information available in completion queue. Again, these actions are part of parametric sub-functions that allow a finer grain of microarchitectural exploration.

The detailed description of generic abstractions for all of the microarchitectural components are too long to describe in this section, and can be found in [49].

4.3 Reference Model Generation

We use the functional abstraction technique to generate executable models (such as a simulator and synthesizable hardware) from the ADL specification. The model generation procedure is same for generating the simulator, hardware (synthesizable RTL), as well as validation models. Only difference is that the input library (consisting of generic functions and sub-functions) needs to be implemented using the appropriate language. For example, the generic functions and sub-functions need to be implemented using programming languages such as C/C++ to enable simulator generation. Similarly, to enable hardware generation the generic library needs to be implemented using a synthesizable subset of VHDL/Verilog. The development of the generic library (consisting of implementation of generic functions and sub-functions) is a one-time activity and independent of the architecture.

The reference model generation process consists of three steps. First, the ADL specification is read to gather all the necessary details for the model generation. Second, the functionality of each component is composed using the generic functions and sub-functions. Finally, the structure of the architecture is composed using the structural details. In the remainder of this section we describe last two steps for simulator generation. As mentioned earlier, the procedure remains the same for generation of hardware and validation models.

Component Generation

To compose the functionality of each component, all necessary details (such as parameters and functionality) are extracted from the ADL specification. First, we describe how to generate three major components of the processor: instruction decoder, execution unit, and controller, using the generic functions and sub-functions. Next, we describe how to compose the functionality of new instructions (behavior) using the generic functions.

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 ADL specification. The decoder extracts information regarding the opcode and operands from the input instruction using the instruction format. The mapping section of the ADL captures 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.

To compose an execution unit, it is necessary to instantiate all the operation functionalities (e.g, ADD, SUB etc. for an ALU) supported by that execution unit. The execution unit invokes the appropriate opcode functionality for an incoming operation based on a simple table look-up technique as shown in Figure 4.7(a). Also, if the execution unit is supposed to read the operands, the appropriate number of operand read functionalities need to be instantiated unless the same read functionality can be shared using multiplexers. Similarly, if the execution unit is supposed to write

```
( TARGET
  ( (MACcc dest src1 src2 src3) )
)
( GENERIC
  ( (MUL temp src1 src2) (ADD dest src3 temp) (RESET CR[2]) )
)
```

Figure 4.6: Mapping between *MACcc* and generic instructions

the data back to register file, the functionality for writing the result needs to be instantiated. The actual implementation of an execute unit might contain many more functionalities such as read latch, write latch, modify reservation station (if applicable), and so on.

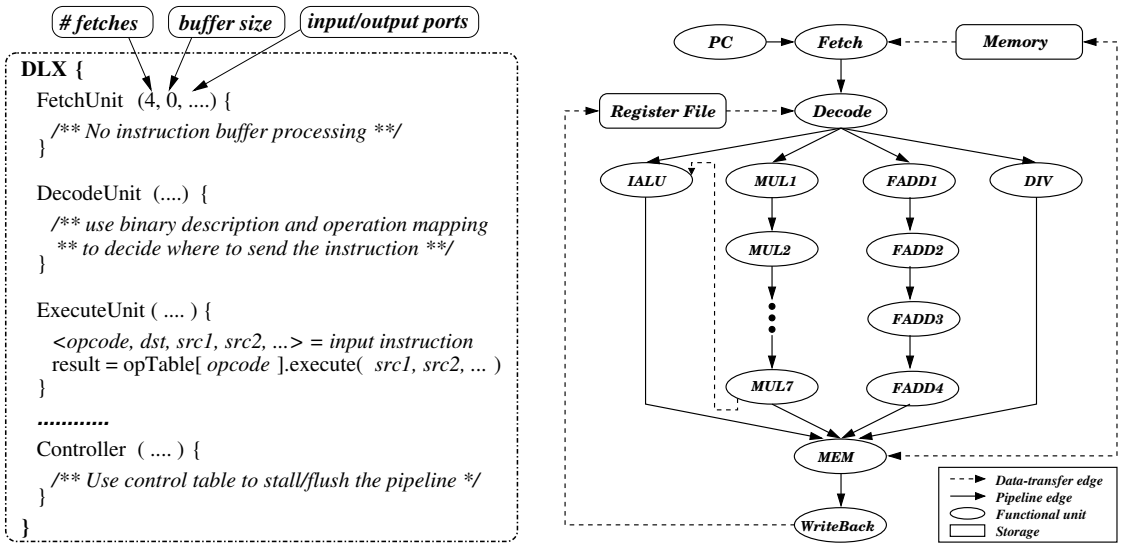
The controller is implemented in two parts. First, it generates a centralized controller (using generic controller function with appropriate parameters) that maintains the information regarding each functional unit, such as busy, stalled etc. It also computes hazard information based on the list of instructions 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 execute 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.

So far we have discussed composition of the structural components for an architecture. It is also necessary to compose the functionality of new instructions (behavior) using the functionality of existing instructions. The EXPRESSION ADL based framework assumes a generic set of instructions (generic architecture). While describing a new architecture (target architecture) using the ADL, it is necessary to provide the mapping between target instructions and generic instructions. This instruction mapping information is typically used by a compiler during instruction selection. This mapping is also used to generate the functionality for the target (new) instructions using the the functionality of the corresponding generic instructions. The scheme

allows one-to-many, many-to-one, and many-to-many mappings between generic and target instructions. For example, the *MACcc* instruction shown in Figure 4.6 uses three generic instructions. The first two generic instructions perform the multiply and accumulate. The third instruction clears the carry bit of the control register.

Processor Model Generation

The final implementation is generated by instantiating components (e.g., fetch, decode, ALU, LdSt, writeback, branch, caches, register files, memories etc.) with appropriate parameters and connecting them using the information available in the ADL. For example, Figure 4.7(a) shows a portion of the simulation model for the DLX architecture shown in Figure 4.7(b).



(a) Simulation model (b) The DLX architecture

Figure 4.7: Simulation model generation for the DLX architecture

The generated simulation models combined with the existing simulation kernel creates a cycle-accurate structural simulator that executes the assembly instructions. In our framework, the assembly instructions generated by the EXPRESS compiler [21] are loaded in the instruction memory of the simulator.

We have used the generated reference models in three top-down validation scenarios: design space exploration, design validation, and test generation. Section 4.4 presents exploration experiments using generated simulator and hardware models. Chapter 5 describes design validation using equivalence checking between the implementation and the generated hardware model. Finally, Chapter 6 presents test generation techniques using the generated validation models.

4.4 Design Space Exploration

Embedded systems present a tremendous opportunity to customize designs by exploiting the application behavior. Specification-driven simulator and hardware generation enable exploration of programmable architectures for a given set of application programs under various design constraints such as area, power and performance. Figure 4.8 shows our architectural exploration framework. The application programs are compiled using the EXPRESS compiler [21] and simulated using the generated simulator, and the feedback is used to modify the ADL specification. Similarly, the generated hardware is used to perform exploration based on silicon area, power, and clock frequency. First, we present exploration experiments using generated simulators. Next, we present exploration experiments using generated hardware models.

4.4.1 Simulator Generation and Exploration

Simulation is the most widely used means of architecture validation. Instruction-set architecture (ISA) simulators are an integral part of todays processor and software design process. While increasing complexity of the architectures demands high performance simulation, the increasing variety of available architectures makes retargetability a critical feature of an instruction-set simulator. We have performed exploration by varying different architectural features. In this section we illustrate the usefulness of our approach in three architecture exploration dimensions.

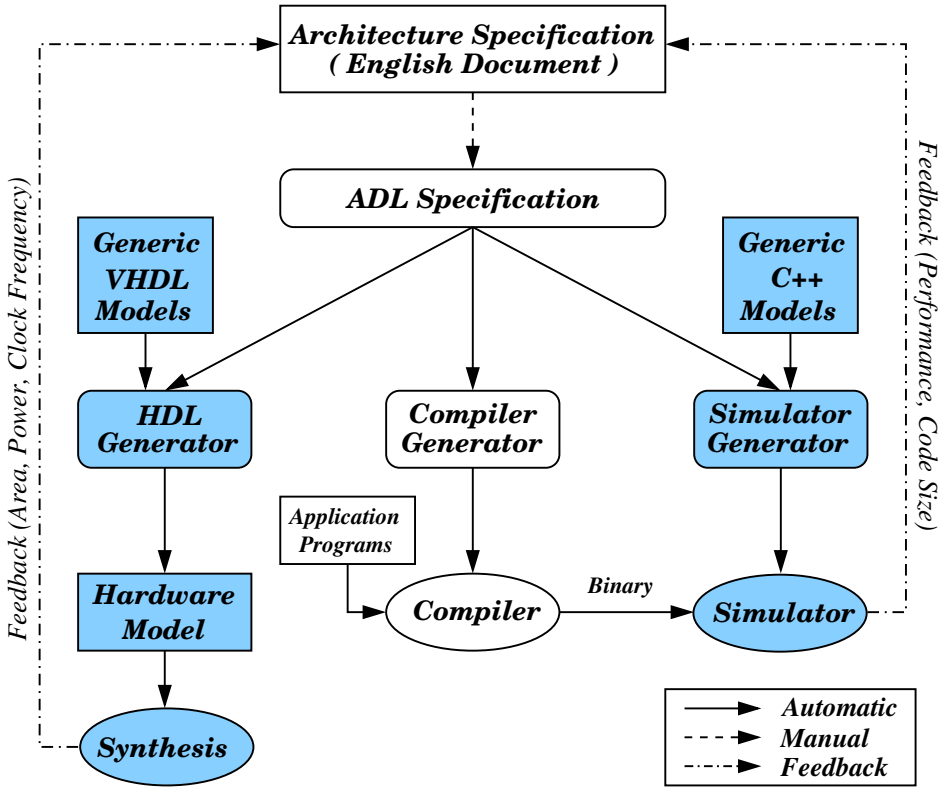


Figure 4.8: Architecture exploration framework

A. Exploration varying Processor Features

Contemporary superscalar processors use in-order completion (graduation) to ensure sequential execution behavior in the presence of out-of-order execution. Here, we explore the MIPS R10K processor in the presence of out-of-order graduation without violating functional correctness. We used a simplified version of the MIPS R10K architecture. It includes the pipeline level description of the processor structure and memory subsystem [59]. We modeled the MIPS R10K architecture (with in-order graduation and an eight entry Active List) using functional abstraction and generated the software toolkit from the specification. We modified the description to perform out-of-order graduation and generated the simulator. We used a set of benchmarks from the multimedia and DSP domains for the experiments.

Figure 4.9 presents a subset of the experiments we ran to study the performance improvement due to out-of-order graduation. The light bar presents the number

of execution cycles when in-order graduation is used whereas the dark bar presents the number of execution cycles when out-of-order graduation is used. We observe an average performance improvement of 10%. During in-order graduation certain instructions (independent of the instructions above in the Active List) complete execution but are not allowed to graduate since some long latency operations are on top of the Active List (completion queue) and are yet to complete. As a result, the Active List becomes full and the decode stalls. This situation becomes more prominent when the top instruction is a load and the load misses. We modified the memory subsystem to study the impact of cache misses along with out-of-order graduation and observed upto 27% performance improvement (in benchmark StateExcerpt when hit ratio is zero). The complete study of the out-of-order graduation for the MIPS R10K processor is available in [59].

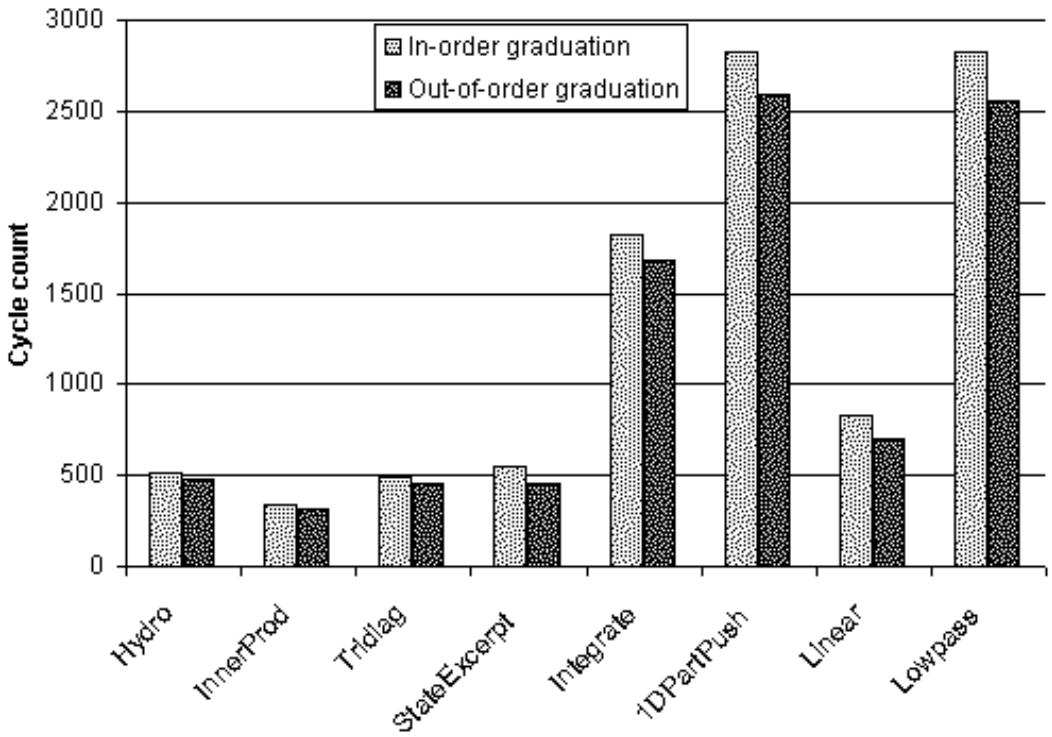


Figure 4.9: Cycle counts for different graduation styles

Due to the high modeling efficiency of functional abstraction, the original description and toolkit generation took less than a week; the graduation style modification

and toolkit generation took less than a day; the experiments and analysis took few hours; the complete exploration experiment took approximately one week.

B. Co-processor based Exploration

In the context of co-processor codesign for programmable architectures we explored the performance impact of using a co-processor for the TI C62x architecture. We used a simplified version of the TI C62x architecture. It includes the pipeline level description of the processor structure and memory subsystem [67]. First, we modeled the TI C62x architecture (where multiplication is done in the functional unit) using functional abstraction and generated the software toolkit. Next, we modified the description by adding a co-processor that supports multiplication and generated the simulator. This co-processor has its own local memory and uses DMA to transfer data from main memory. We used a set of DSPStone fixed point benchmarks to explore and evaluate the effects of adding a coprocessor.

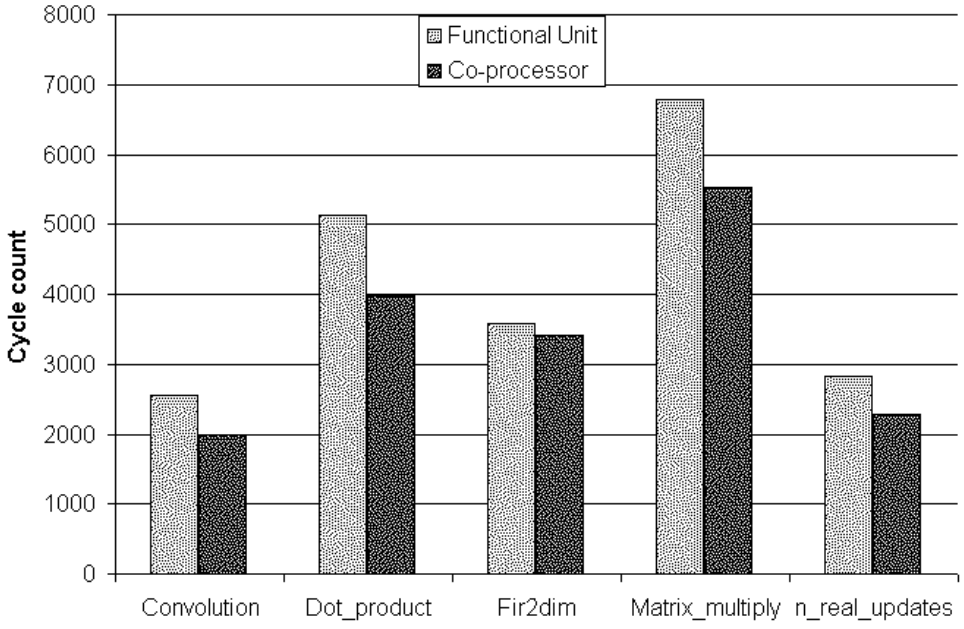


Figure 4.10: Functional unit vs. coprocessor

Figure 4.10 presents a subset of the experiments we ran to study the performance improvement due to the co-processor. The light bar presents the number of execution

cycles when the functional unit is used for the multiplication whereas the dark bar presents the number of execution cycles when the co-processor is used. We observe an average performance improvement of 22%. The performance improvement is due to the fact that the co-processor is able to exploit the vector multiplications available in these benchmarks using its local memory. Moreover, functional units operate in register-to-register mode whereas co-processor operates on its memory-memory mode. As a result the register pressure and thereby spilling gets reduced in the presence of the co-processor. However, the functional unit performs better when there are mostly scalar multiplications. The complete study of the co-processor based design space exploration is available in [67].

C. Processor-Memory Co-Exploration

While a traditional memory architecture for programmable systems was organized as a cache hierarchy, the widening processor/memory performance gap [78] requires more aggressive use of memory configurations, customized for specific target applications. To address this problem, recent advances in memory technology have generated a plethora of new and efficient memory modules (e.g., SDRAM, DDRAM and RAMBUS), exhibiting a heterogeneous set of features (e.g., page-mode, burst-mode and pipelined accesses). On the other hand, many embedded applications exhibit varied memory access patterns that naturally map into a range of heterogeneous memory configurations (containing for instance multiple cache hierarchies, stream buffers, on-chip and off-chip direct mapped memories). Due to the heterogeneity in recent memory organizations and modules, there is a critical need to address the memory-related optimizations simultaneously with the processor architecture and the target application.

We present a set of experiments using our memory-aware ADL to drive the exploration of the memory subsystem for the TI C6211 processor architecture, demonstrating cost, performance, and energy trade-offs. First, we describe the experimental setup. Next, we present the exploration results.

Experimental Setup

We performed a set of experiments starting from the base TI C6211 [89] processor architecture, and varied the memory subsystem architecture. We generated a memory-aware software toolkit (compiler and simulator), and performed design space exploration of the memory subsystem. The memory organization of the TIC6211 is varied by using separate L1 instruction and data caches, an L2 cache, an off-chip DRAM module, an on-chip SRAM module, and a stream buffer module [38] with varied connectivity among these modules.

Table 4.3: Benchmarks

Benchmark	Description
Compress	Image compression scheme
GSR	Red-black Gauss-Seidel relaxation method
Hydro	Hydro fragment
DiffPred	Difference predictors
FirstSum	First sum
FirstDiff	First difference
PartPush	2-D PIC (Particle In Cell)
1DPartPush	1-D PIC (Particle In Cell)
CondCompute	Implicit, conditional computation
Hydrodynamics	2-D explicit hydrodynamics fragment
GLRE	General linear recurrence equations
ICCG	ICCG excerpt (Incomplete Cholesky Conjugate Gradient)
MatMult	Matrix multiplication
Planc	Planckian distribution
2DHydro	2-D implicit hydrodynamics fragment
FirstMin	Find location of first minimum in array
InnerProd	Inner product
LinearEqn	Banded linear equations
TriDiag	Tri-diagonal elimination, below diagonal
Recurrence	General linear recurrence equations
StateExcerpt	Equation of state fragment
Integrate	ADI integration
IntPred	Integrate predictors
Laplace	Laplace algorithm to perform edge enhancement
Linear	Implements a general linear recurrence solver
Wavelet	Debaucles 4-Coefficient Wavelet filter

We used benchmarks from the multimedia and DSP domains for our experiments. The list of the benchmarks is shown in Table 4.3. The benchmarks are compiled using the EXPRESS compiler [21]. We collected the statistics information using the generated simulator that models both the TIC6211 processor and the memory subsystem.

Table 4.4: The memory subsystem configurations

Cfg	Area (rbe)	L1 ICache (latency=1)	L1 DCache (latency=1)	L2 Cache (latency=5)	SRAM (lat=1)	Str. Buffer (lat=5)
1	54567	256B (8x2x4x4)	256B (8x2x4x4)	8K (256x8x1x4)	-	-
2	61335	256B (8x2x4x4)	256B (8x2x4x4)	4K (64x2x8x4)	4K	-
3	60449	256B (8x2x2x4)	256B (8x2x4x4)	8K (64x4x8x4)	-	-
4	66394	256B (8x2x4x4)	256B (8x2x4x4)	8K (64x4x8x4)	-	8x2x8x4
5	125466	256B (8x2x4x4)	256B (8x2x4x4)	2K (16x4x8x4)	16K	8x2x8x4
6	51169	128B (8x2x2x4)	128B (8x2x2x4)	8k (256x8x1x4)	-	-
7	52868	128B (8x2x2x4)	256B (8x2x4x4)	8k (256x8x1x4)	-	-
8	58198	128B (8x2x2x4)	256B (8x4x2x4)	8k (64x4x8x4)	-	-
9	52057	128B (8x2x2x4)	256B (16x2x2x4)	8k (256x8x1x4)	-	-
10	52868	256B (8x2x4x4)	128B (8x2x2x4)	8k (256x8x1x4)	-	-
11	33099	256B (8x4x2x4)	256B (8x4x2x4)	4k (64x4x4x4)	-	-
12	31698	256B (16x2x2x4)	256B (16x2x2x4)	4k (256x4x1x4)	-	-
13	33469	256B (16x2x2x4)	512B (32x2x2x4)	4k (256x4x1x4)	-	-
14	53847	512B (16x4x2x4)	128B (8x2x2x4)	8k (256x8x1x4)	-	-
15	33488	512B (16x4x2x4)	256B (16x2x2x4)	4k (256x4x1x4)	-	-
16	35259	512B (16x4x2x4)	512B (32x2x2x4)	4k (256x4x1x4)	-	-
17	58100	512B (8x8x2x4)	512B (8x8x2x4)	8k (256x8x1x4)	-	-
18	36066	512B (8x8x2x4)	512B (16x4x2x4)	4k (256x4x1x4)	-	-
19	59156	512B (8x4x4x4)	512B (8x4x4x4)	8k (256x8x1x4)	-	-
20	55182	256B (16x2x2x4)	256B (16x2x2x4)	4k (256x4x1x4)	4k	-
21	53406	128B (8x2x2x4)	128B (8x2x2x4)	4k (256x4x1x4)	4k	-
22	76753	128B (8x2x2x4)	128B (8x2x2x4)	4k (256x4x1x4)	8k	-
23	36227	128B (8x2x2x4)	256B (16x2x2x4)	4k (256x4x1x4)	-	8x8x2x4
24	33909	128B (8x2x2x4)	256B (16x2x2x4)	8k (256x4x1x4)	-	8x4x2x4
25	106722	128B (8x2x2x4)	256B (16x2x2x4)	8k (64x8x4x4)	8k	8x8x2x4
26	106722	128B (8x2x2x4)	256B (16x2x2x4)	8k (64x8x4x4)	8k	8x8x2x4
27	85146	128B (8x2x2x4)	512B (32x2x2x4)	8k (64x8x4x4)	4k	8x8x2x4

While any micro-architectural estimation models can be used in our framework, we use area models from Mulder et al. [71] and energy models[2] from Wattch [8]. The

[2]This considers only dynamic power dissipation; leakage effects and static power dissipation have not considered in this study.

performance of a particular memory configuration for a given application program is computed as the number of clock cycles it takes to execute the application in the simulator. We divide this cycle count by 2000 to show both energy and performance plots in the same figure.

Some of the configurations we experimented with are presented in Table 4.4. Each row of the table corresponds to a memory configuration. The second column presents the area of the memory configurations. The remaining entries in the table represent the size of the memory module (e.g., the size of L1 in configuration 1 is 256 bytes) and the cache/stream buffer organizations: $num\_lines \times line\_size \times num\_ways \times word\_size$. The LRU cache replacement policy is used. The latency is defined in number of processor cycles. Note that, for stream buffer the *num_ways* represents the number of FIFO queues present in it. The first configuration contains an L1 instruction cache (256 bytes), L1 data cache (256 bytes), and a unified L2 cache (8K bytes). All the configurations contain the same off-chip DRAM module with a latency of 50 cycles. The cache sizes are decided based on the application programs. We have used benchmarks from the multimedia and DSP domains for our experiments. These benchmarks are small/medium size kernels. Therefore, the cache sizes are smaller than typical sizes used in conventional processor architectures.

Results

Here we analyze a subset of the experiments we ran with the goal of evaluating different memory configurations for area, energy and performance. Figure 4.11 shows the exploration result for the *GSR* benchmark. The X-axis represents the memory configurations in the increasing order of cost in terms of area. The Y-axis represents values for both performance and energy. The performance value is normalized by dividing cycle count by 2000. The energy value is given in μJ. Although the cost for memory configurations 6 and 9 are much lower than the cost of configuration 5, the former (6 and 9) configurations deliver better results in terms of energy and performance. Configuration 21 consumes lower energy and delivers better performance than configuration 6. However, the former is worse than the latter in terms

of area. Depending on the priority among area, energy and performance, one of the configurations can be selected.

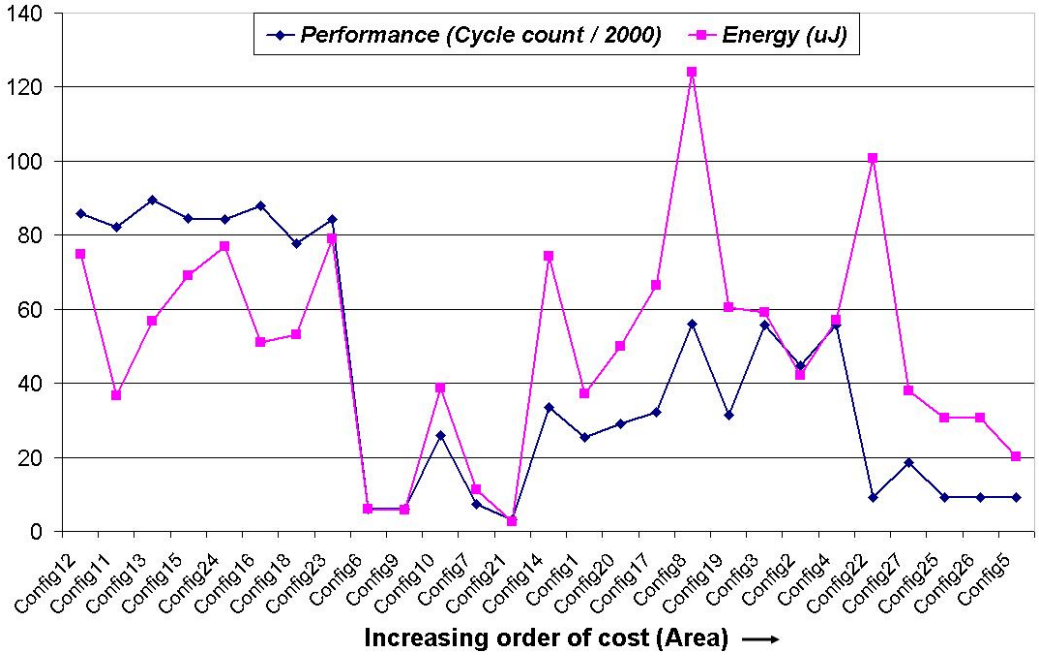


Figure 4.11: Memory exploration results for GSR

When area consideration is not very important we can view the pareto-optimal configurations from energy-performance trade-offs. Figure 4.12 shows the energy-performance trade-off for the *Compress* benchmark. It is interesting to note that a set a memory configurations (with varied parameters, modules and connectivity) deliver similar performance results for the *Compress* benchmark. There are three distinct performance zones. The first zone has performance values between 5 and 10. This zone consists of memory configurations 2, 5, 20, 21, 22, 25, 26, and 27. The energy values are different due to the fact that each configuration has different parameters, modules, connectivity and area. However, the performance is almost similar since the data fits in SRAM of size 2K for these configurations. Similarly, the second zone (configurations 1, 6, 7, 9, 10, 14, 17, 19) has performance values between 15 and 20 with very different power values. The performance is almost same for these configurations because the L2 cache size of 8K or larger has very high hit ratio and as

a result for all these memory configurations L2 dominates and L2 to DRAM access remains almost constant.

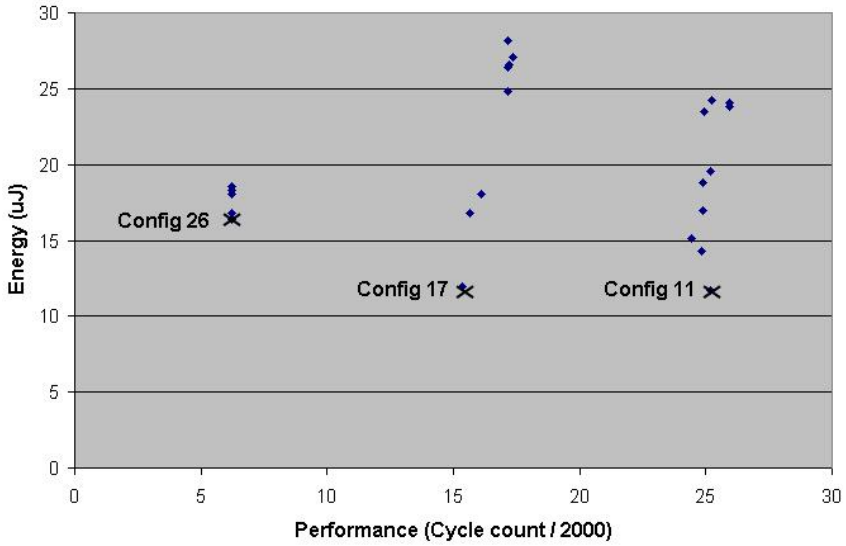


Figure 4.12: Energy performance tradeoff for Compress

Similarly, the third zone (configurations 3, 4, 8, 11, 12, 13, 15, 16, 18, 23, 24) has almost same performance with different power values. This is due to the fact that each of these configurations has L2 line size of 4 that dominates over other parameters for these configurations. This line size is the reason why configurations in the third zone are worse than the configurations in the second zone. Depending on the priority among cost, energy and performance one of the three configurations (Config 11, 17, 26) can be chosen. The same phenomenon can be observed in the benchmarks *FirstSum*, *FirstDiff*, and *FirstMin* [64]. The benchmark *InnerProd* has four such performance zones whereas the benchmark *Tridiag* has five such performance zones [64]. The pareto-optimal configurations are shown using symbol *X*, and the corresponding memory configuration is mentioned in the figure.

However, for some set of benchmarks the energy performance tradeoff points are scattered in the design space and thus only the pareto-optimal configurations are of interest. Figure 4.13 shows the energy performance tradeoff for the benchmark *Mat-Mult*. It has only one pareto-optimal point i.e., configuration 5. However, the *Laplace*

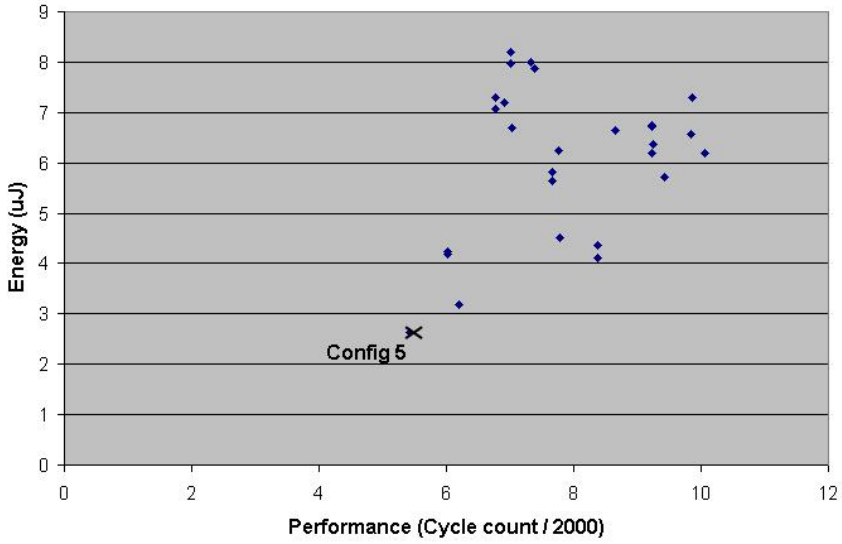


Figure 4.13: Energy performance tradeoff for MatMult

benchmark (Figure 4.14) has two pareto-optimal points: configuration 5 delivers better performance than configuration 17 but consumes more energy and has a larger area requirement. Depending on the priority among area, energy and performance, one of the two configurations can be selected. The energy-performance tradeoff results for the remaining benchmarks are available in [64].

Thus, using our memory-aware ADL based design space exploration approach, we have obtained design points with varying cost, energy and performance. We have observed various trends for different application classes, allowing customization of the memory architectures tuned to the applications. Note that, this cannot be determined through analysis alone; the customized memory subsystem must be explicitly captured, a memory-aware compiler and simulator should be automatically generated, and the applications have to be executed on the configured processor-memory system, as demonstrated in this section.

We have also performed microarchitectural exploration of the MIPS 4000 processor [48] in three directions: pipeline exploration, instruction-set exploration, and memory exploration [72]. Pipeline exploration allows the addition (deletion) of new (existing) functional units or pipeline stages. Instruction-set exploration allows addition of new

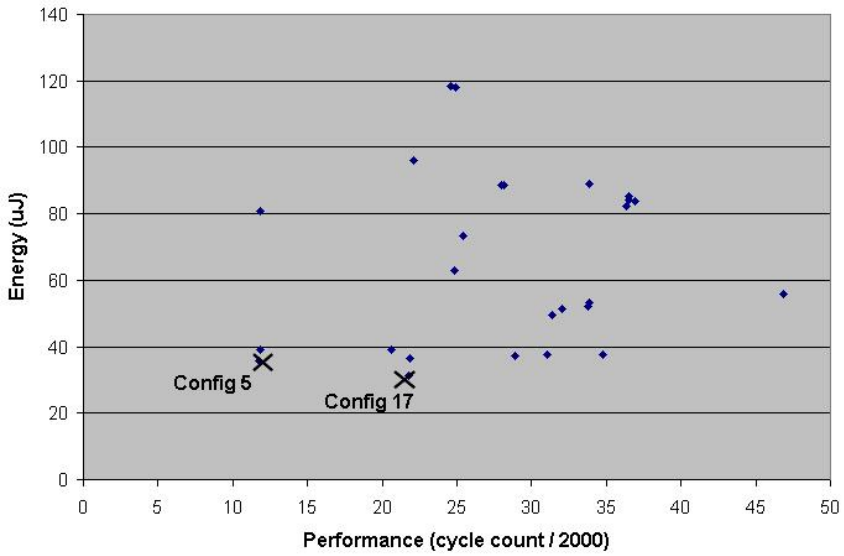


Figure 4.14: Energy performance tradeoff for Laplace

instructions or formation of complex instructions by combining the existing instructions. Similarly, memory exploration allows modification of memory hierarchies and cache parameters. The user can modify the ADL specification of the architecture, and the software toolkit, including a compiler and a simulator, is automatically generated from the ADL specification. The public release of the retargetable simulation and exploration framework is available from *http://www.cecs.uci.edu/~express*.

4.4.2 Hardware Generation and Exploration

The simulator produces profiling data and thus may answer questions concerning the instruction set, the performance of an algorithm and the required size of memory and registers. However, the required silicon area, clock frequency, and power consumption can only be determined in conjunction with a hardware model. In this section, we illustrate the use of ADL-driven synthesizable hardware model generation for exploration of the DLX [22] processor by varying different architectural features.

Figure 3.6 shows the pipelined DLX architecture. The EXPRESSION ADL captures the structure and behavior of the DLX architecture. Synthesizable HDL models are generated automatically from the ADL specification using the procedure described

in Section 4.3. We have used Synopsys Design Compiler [84] to synthesize the generated HDL description using LSI 10K technology library and obtained area, power and clock frequency values.

```
arg = th2 * piovn
c1 = cos(arg)
s1 = sin(arg)
c2 = c1 * c1 - s1 * s1;
s2 = c1 * s1 + c1 * s1;
c3 = c1 * c2 - s1 * s2;
s3 = c2 * s1 + s2 * c1;

int4 = in * 4;
j0 = jr * int4 + 1;
k0 = ji * int4 + 1;
jlast = j0 + in - 1;
```

Figure 4.15: The application program

Figure 4.15 shows one of the most frequently executed code segment from FFT benchmark that we have used as an application program during micro-architectural exploration.

We have performed architectural explorations on this DLX model by varying different micro-architectural features [39]. In this section we present three exploration experiments: pipeline path exploration, pipeline stage exploration, and instruction-set exploration. The reported area, power, and clock frequency numbers are for the execution units only. The numbers do not include the contributions from others components such as *Fetch*, *Decode*, *MEM* and *WriteBack*.

Addition of Functional Units (Pipeline Paths)

Figure 4.16 shows the exploration results due to addition of pipeline paths using the application program shown in Figure 4.15. The first configuration has only one pipeline path consisting of *Fetch*, *Decode*, one execution unit (say *Ex1*), *MEM* and *WriteBack*. The *Ex1* unit supports five operations: *sin*, *cos*, +, – and ×. The second configuration is exactly same as the first configuration except it has one more execution unit (say *Ex2*) parallel to *Ex1*. The *Ex2* unit supports three operations: +, – and ×. Similarly, the third configuration has three parallel execution units: *Ex1* (+, –, ×), *Ex2* (+, –, ×), and *Ex3* (*sin*, *cos*, +, – and ×). Finally, the fourth configuration has four parallel execution units: *Ex1* (sin, cos), *Ex2* (+, –, MAC[3]), *Ex3*, and *Ex4*,

[3]MAC performs multiply-and-accumulate of the form $a \times b + c$

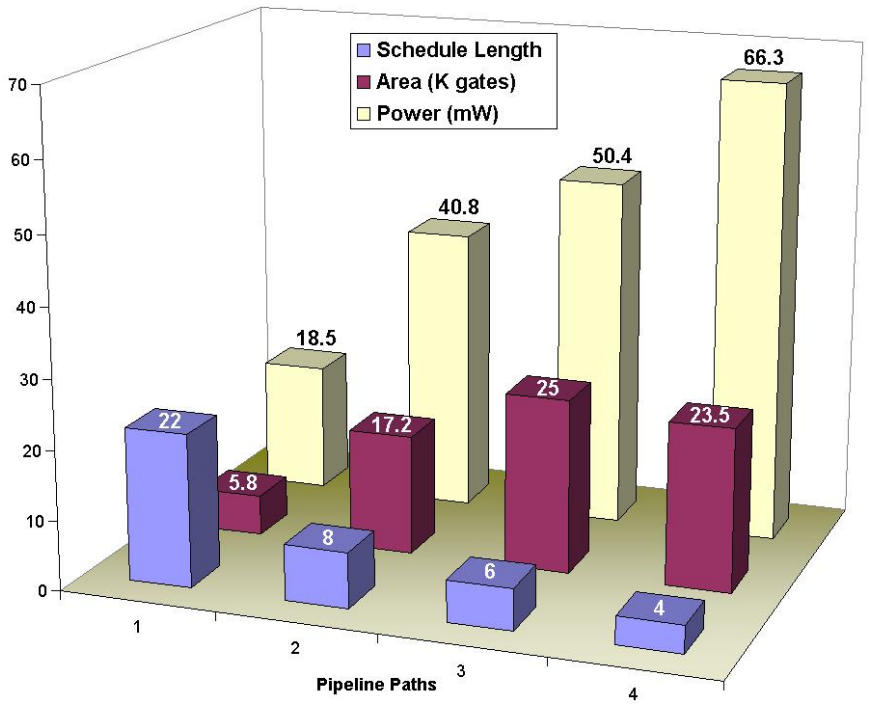


Figure 4.16: Pipeline path exploration

where *Ex3* and *Ex4* are customized functional units that perform $a \times b + c \times d$.

As expected, the application program requires fewer number of cycles (schedule length) due to addition of pipeline paths whereas the area and power requirement increases. The fourth configuration is interesting since both area and schedule length decrease due to addition of specialized hardware and removal of operations from other execution units.

Addition of Pipeline Stages

Figure 4.17 presents exploration experiments due to addition of pipeline stages in the multiplier unit. The first configuration is a one-stage multi-cycle multiplier. The second, third and fourth configurations use multipliers with two, three and four stages respectively.

As expected, the clock frequency (speed) is improved due to addition of pipeline stages. The fourth configuration generated 30% speed improvement at the cost of

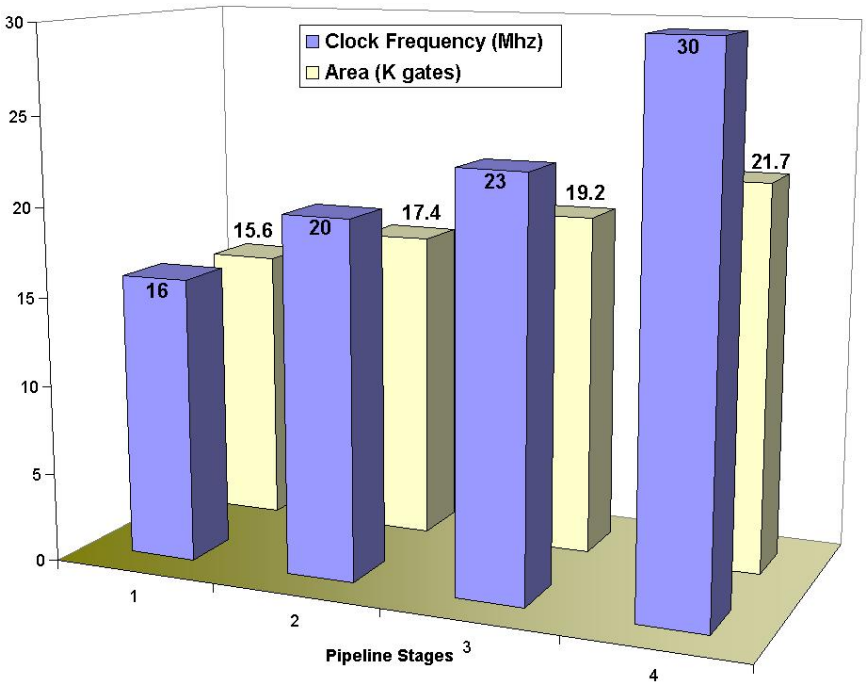


Figure 4.17: Pipeline stage exploration

13% area increase over the third configuration. The designer needs to decide whether 13% area increase is acceptable for the 30% speed improvement.

Addition of Operations

Figure 4.18 presents exploration results for addition of opcodes using three processor configurations. The three configurations are shown in Figure 4.18. The first configuration has four parallel execution units: *FU1*, *FU2*, *FU3* and *FU4*. The *FU1* supports three operations: $+$, $-$, and $\times$. The *FU2*, *FU3* and *FU4* supports ($+$, $-$, $\times$), (*and*, *or*), and (*sin*, *cos*) respectively. The second configuration is obtained by adding a *cos* operation in the *FU3* of the first configuration.

As expected, this generated reduction of schedule length of the application program at the cost of area increase. The third configuration is obtained by adding multipliers both in *FU3* and *FU4* of the second configuration. This generated best possible (using $+$, $-$, $\times$, *sin* and *cos*) schedule length for the application program shown in Figure 4.15.

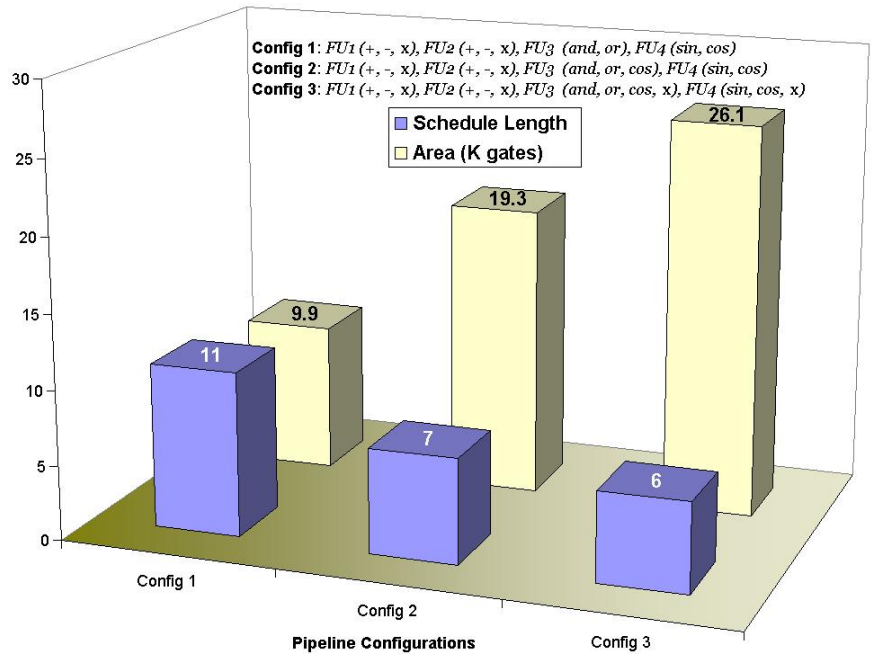


Figure 4.18: Instruction-set exploration

Each iteration in our exploration framework is in the order of hours to days depending on the amount of modification needed in the ADL and the synthesis time. However, each iteration will be in the order of weeks to months for manual or semi-automatic development of HDL models. The reduction of exploration time is at least an order of magnitude.

4.5 Chapter Summary

A major challenge in top-down validation methodology is the ability to generate executable models from the specification for a wide variety of programmable architectures including RISC, DSP, VLIW, and superscalar. We have studied the similarities and differences of each architectural feature in different architecture domains. Based on our observations we have defined generic functions, sub-functions, and computational environment needed to capture a wide variety of programmable architectures. Our functional abstraction technique enables model generation for simulation, hard-

ware generation, and property checking from the ADL specification. The generated models are used for design validation, test generation, and design space exploration.

The second part of this chapter presented exploration experiments for programmable architectures for a given set of application programs under various design constraints such as area, power, and performance. We presented exploration results using generated simulation models in the context of three scenarios: exploration varying processor features, coprocessor exploration, and processor-memory co-exploration. We also presented results using generated hardware models in three exploration scenarios: pipeline path exploration, pipeline stage exploration, and instruction-set exploration. We have obtained design points with varying cost, energy, and performance attributes using ADL-driven design space exploration.

Chapter 5

Specification-driven Validation

One of the major challenges in validation of programmable embedded systems is the verification of RTL design (implementation). Design validation techniques can be broadly categorized into simulation-based approaches and formal techniques. Due to the complexity of modern designs, validation using only traditional scalar simulation cannot be exhaustive. Formal techniques exhaustively analyze parts of the design but, because of state space explosion, are not suitable for the complete design. *Equivalence Checking* is one of the most widely used formal techniques in industry today. Typically, the implementation is compared with a set of Boolean equations, or an optimized circuit is compared with the original circuit. *Symbolic simulation* has proven to be an efficient technique, bridging the gap between traditional simulation and full-fledged formal verification.

Figure 1.2 shows a traditional architecture validation flow. The implementation design is validated using a combination of simulation techniques and formal methods. The existing techniques employ a bottom-up approach to validation, where the functionality of an existing processor is, in essence, reverse-engineered from its RTL implementation. The validation technique presented in this thesis is complementary to these bottom-up approaches. This chapter presents a top-down methodology for validation of microprocessors.

Figure 5.1 shows our validation methodology. Logic designers implement the architecture at register-transfer level (*RTL design*). The structure and behavior of the

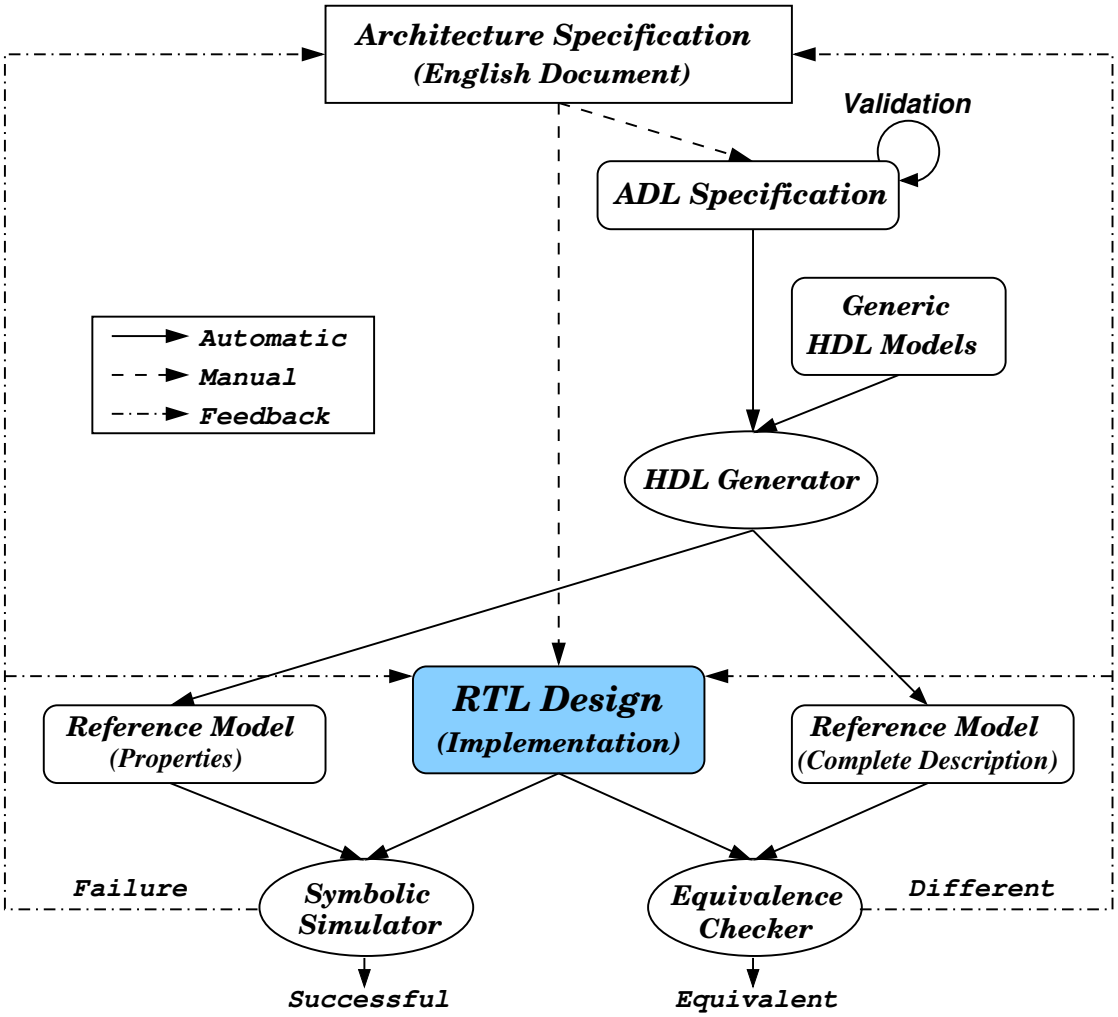


Figure 5.1: Top-down validation methodology

processor is captured using an ADL. Chapter 2 describes how to specify architectures using the EXPRESSION ADL. The ADL specification is validated to ensure that it specifies a well-formed architecture using the techniques presented in Chapter 3. The reference model (HDL description) is automatically generated from the ADL specification using the procedure described in Section 4.3.

Our validation framework allows generation of synthesizable RTL description of the architecture as well as specific properties. The RTL description can be used for checking equivalence with the given implementation. However, generation of specific behaviors would enable property checking. For example, our framework generates

the property: $output = \sum_{i=1}^{n} input_i$, for a n-input adder. The design should satisfy this property irrespective of the adder implementation, such as ripple carry adder or carry lookahead adder.

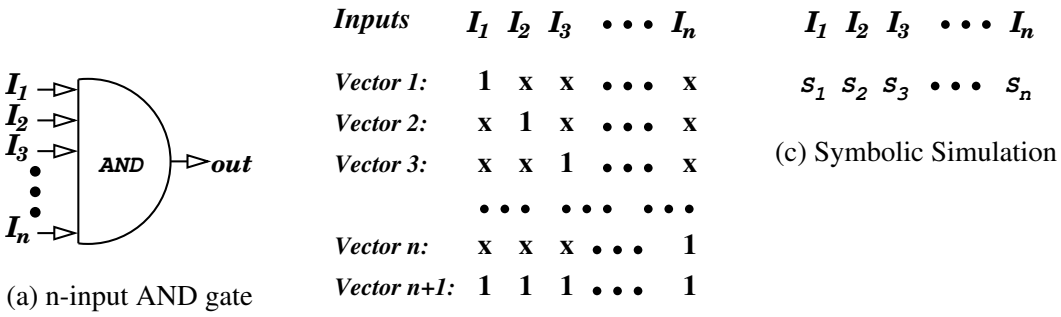
A major advantage of property checking is that it reduces the complexity of verification. However, this technique raises an important question: how to choose the set of properties. A set of properties can be chosen in two different ways. First, the designers can decide what properties are important to be verified for the design based on their design knowledge and past experience. They can then choose the properties to uncover otherwise difficult-to-find bugs. Second, a set of behaviors can be chosen and their effectiveness can be evaluated. For example, to verify a memory controller in a microprocessor, it is necessary to generate properties to validate each output of the controller. To measure the effectiveness of these properties, a set of coverage measures can be used during property checking [13].

We use the Versys2 symbolic simulator [41] to perform property checking. A counter-example is generated if a property fails in the *RTL design*. The feedback is used to modify the *RTL design*. Our framework uses Synopsys Formality [85] to perform equivalence checking between the implementation and the generated RTL description. In case of a failure, the feedback is used to modify the *RTL design*. If there is an ambiguity in the original description that led to the mismatch, the architecture specification needs to be updated.

This chapter is organized as follows. Section 5.1 describes our validation methodology using a combination of symbolic simulation and equivalence checking. Section 5.2 presents the validation experiments. Finally, Section 5.3 summarizes the chapter.

5.1 Design Validation

Our top-down validation methodology has the ability to perform both model (property) checking and equivalence checking depending on the generated reference model. This section describes the validation techniques used in our framework.



(a) n-input AND gate

(b) Ternary Simulation

(c) Symbolic Simulation

Figure 5.2: Test vectors for validation of an *AND* gate

5.1.1 Property Checking using Symbolic Simulation

Symbolic simulation combines traditional simulation with formal symbolic manipulation [9]. Each symbolic value represents a signal value for different operating conditions, parameterized in terms of a set of symbolic Boolean variables. By this encoding, a single symbolic simulation run can cover many conditions that would require multiple runs of a traditional simulator. Figure 5.2(a) shows a simple n-input *AND* gate. Exhaustive simulation of the *AND* gate requires 2^n binary test vectors. However, the ternary simulation (uses 0, 1, and x) requires $(n + 1)$ test vectors for the *AND* gate. Figure 5.2(b) shows the vectors: n vectors with one input set to '1' and the remaining inputs set to 'x', and one vector with all inputs set to '1'. Finally, symbolic simulation [9] requires only one vector using n symbols (s_1, s_2, ..., s_n) as shown in Figure 5.2(c).

Researchers at IBM first introduced symbolic simulation to reason about properties of circuits described at the register-transfer level. With the advent of Binary Decision Diagrams (BDDs), the technique became much more practical. Providing a canonical representation for Boolean functions, BDDs enabled the implementation of an efficient event-driven logic simulator that operated over a symbolic domain. By encoding a model's finite domain using a Boolean encoding, it is possible to symbolically simulate the model using BDDs. Bryant's formal state transition model for a ternary system [10], and Seger's work on symbolic trajectory evaluation renewed further interest in symbolic execution [76].

The symbolic simulator (used in our framework) uses symbolic trajectory evaluation (STE). In this section we informally describe STE. The formal description of STE is available in [76]. STE is a modified form of symbolic simulation that operates over the quaternary logic domain 0, 1, X, and T [76]. A state of the circuit is defined as the set of all node values at a particular time instant. The value domain is partially ordered and forms a complete lattice, $X \sqsubseteq 0$ indicates X has less information than 0, or X is weaker than 0. The information content of 0 and 1 are not comparable. If $r \sqsubseteq q$ and $r \sqsubseteq t$, we can think of r as representing both q and t. Any property that holds for a state such as r will also hold for all the states above it in the lattice, for example q and t.

STE provides a mathematically rigorous method for establishing that properties (assertions) of the form *antecedent (A)* $\Rightarrow$ *consequent (C)* hold for a given simulation model of a circuit. For the test vector shown in Figure 5.2(c), the antecedent is: (I_1 is s_1, I_2 is s_2, ..., I_n is s_n) from time 0 to 1, and the consequent is: *out* is $s_1 \& s_2 \& ... \& s_n$ from time 1 to 2. Circuit state holders are initialized with symbolic values specified by the antecedent. The model is then simulated, typically for one or two clock cycles, while driving the inputs with symbolic values during simulation. The resulting values, appear on selected internal nodes and primary outputs, are compared with the expected values expressed in the consequent. In general, the values could be functions over a finite set of variables. A trajectory is a sequence of states such that each state has at least as much information as the next-state function applied to the previous state. Intuitively, a trajectory is a state sequence constrained by the system's next-state function. A successful simulation of assertion $A \Rightarrow C$ establishes that any sequence of assignments of values to circuit nodes that is both consistent with the circuit behavior and consistent with antecedent A is also consistent with consequent C.

Symbolic trajectory evaluation is used to verify that an implementation satisfies its specification (reference model). Necessary assertions are extracted from the reference model. If the implementation (e.g., *RTL design*) is correct, these assertions should hold during symbolic simulation of the *RTL design*. An assertion ($A \Rightarrow C$) holds if the weakest antecedent trajectory that the implementation goes through during

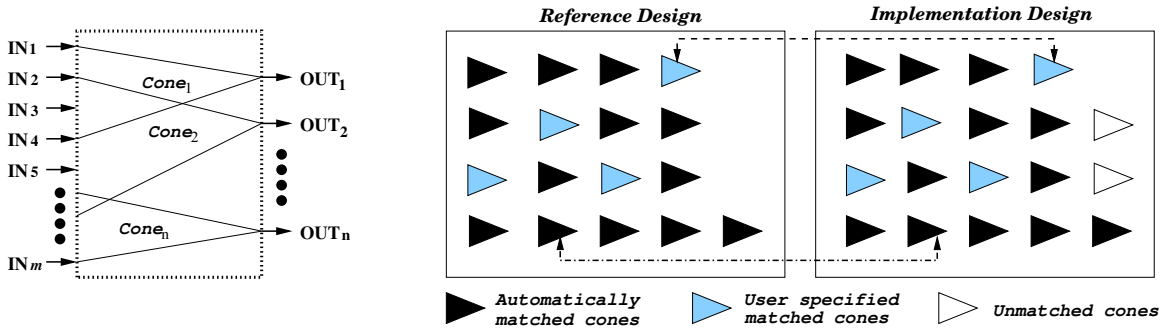
simulation (using A) should be at least as strong as the weakest sequence satisfying the consequent C. Informally, the outputs produced during simulation (using A) should be at least as strong as the expected outputs (given in C).

To verify that the implementation satisfies certain properties, our framework generates the intended properties instead of generating the complete reference design. We use the Versys2 [41] that uses symbolic trajectory evaluation to perform property checking. It is necessary to manually specify the state mappings between the reference model and the implementation. This involves mapping of both latches and bit cells by specifying their names. The assertions are automatically generated from the reference model [94]. Versys2 symbolically simulates the implementation by using the generated assertions to ensure that the implementation satisfies the reference model. A counter-example is generated if an assertion fails in the implementation. The feedback is used to modify the implementation.

5.1.2 Equivalence Checking

Equivalence Checking is a branch of static verification that employs formal techniques to prove that two versions of a design either are, or are not, functionally equivalent. The equivalence checking flow consists of four stages: *reading*, *matching*, *verification* and *debugging*. The matching and verification stages are those most impacted by design transformations. During the *reading* stage, both versions of the design are read by the equivalence checking tool and segmented into manageable sections called logic cones. Logic cones are groups of logic bordered by registers, ports, or black boxes. Figure 5.3(a) shows the cones for a typical design block. The output border of a logic cone is referred to as the compare point. For example, OUT_1 is the compare point in $Cone_1$ of Figure 5.3(a).

In *matching* phase, the tool attempts to match, or map, compare points from the reference design to their corresponding compare point within the implementation design [4]. Two types of matching techniques are used: name based (non-function) and function based (signature analysis). Figure 5.3(b) shows compare point matching for a typical reference design and implementation. For better performance, the



a) Logic Cones in a Design Block

b) Compare Point Matching

Figure 5.3: Compare point matching between reference and implementation design

majority of the matching should be completed by more efficient name based methods. Design transformations can result in fewer cones being matched by the name based techniques, slowing match performance. Creating compare rules assist name based techniques, but determination and creation of the rules themselves can be time consuming. If the implementation is drastically different than the reference design, design rules cannot be written and compare points have to be manually matched for better performance or matched using more costly function based techniques. This becomes impractical for design with many unmatched points.

During the *verification* stage, each matched compare point is proven either functionally equivalent or non-equivalent ([16], [46]). Design transformations can impact the structure of a logic cone in the implementation design. When logic cones are very dissimilar, performance suffers. In some cases, such as during retiming, the logic cones can change so significantly that additional setup is required to successfully verify the designs. The *debugging* phase begins when the tool has returned a non-equivalent result. Design transformations that have not been accounted for can lead to a false negative result, and valuable time could be spent debugging designs that are, in reality, equivalent. The solution would be to perform additional setup so that the tool is guided for the given designs.

Our framework generates the synthesizable RTL description of the processor to enable equivalence checking using Synopsys Formality [85]. The tool reads both the reference and the implementation designs, and attempts to match the compare points

between them. The unmatched compare points need to be mapped manually. The tool tries to establish equivalence for each matched compare point. In case of a failure, the failing compare points are analyzed to verify whether they are actual failures or not. The feedback is used to perform additional setup (in case of a false negative), or to modify the implementation.

Specification-driven design validation using equivalence checking has one limitation: the structure of the generated hardware model (reference) needs to be similar to that of the implementation. This requirement is primarily due the limitation of the equivalence checkers available today. Equivalence checking is not possible using these tools if the reference and implementation designs are large and drastically different. As a result, our methodology is applicable when the reference model generation is guided to have a structure similar to the implementation. Section 5.2.2 presents the validation of a RISC DLX processor [22] using equivalence checking.

5.2 Experiments

An important aspect of our methodology is the ability to perform both model (property) checking and equivalence checking depending on the generated reference model. To verify that the implementation satisfies certain properties, our framework generates the intended properties instead of generating the complete reference design. Section 5.2.1 presents validation of a memory management unit of a microprocessor that is compliant with the PowerPC instruction-set using model checking. On the other hand, if the generated reference model contains the RTL description of the design, our framework performs equivalence checking between the implementation and the generated reference model. Section 5.2.2 presents the validation of a RISC DLX processor using equivalence checking.

5.2.1 Property Checking of a Memory Management Unit

The memory management unit (MMU) supports demand-paged virtual memory. It consists of blocks such as *Segment Registers*, *Translation Lookaside Buffers (TLBs)*,

and *Block Address Translation (BAT) Arrays*. Each of these memory blocks are composed of sub-blocks. For example, a TLB has three sub-blocks: *entry* (data information), *LRU* (least recently used information), and *valid* (information regarding validity of the data) as shown in Figure 5.4. Each of these sub-blocks is implemented as SRAM. The typical operations in SRAM are read and write. Therefore, a natural property to verify is to check read and write for each SRAM cell. The generated reference model contains the following Verilog code segment to verify the read and write properties for an SRAM cell.

```
always @ (wrClk or wrEn or dIn or wrAddr)
begin
  if (wrClk & wrEn) ram[wrAddr] <= dIn;
end

assign out = (rdClk & rdEn) ? ram[rdAddr] : 32'b0;
```

The Versys2 symbolic simulator does not have automatic node matching (compare point matching) scheme. Therefore, it is necessary to manually map the nodes between the reference model and the implementation. We modified Versys2 configuration file to provide the node mapping between the reference model and the implementation. For example, the *wrClk* of the reference model is mapped to *sramWrClk* of the implementation. An interesting feature of this validation approach is that the same set of properties (without any modification) is applied to all MMU memory blocks. However, in each case, the node mapping must be modified.

To verify whether the *RTL design* correctly implements the TLB miss detection, our framework generated the following Verilog code segment. The information needed to build this property is directly available from the specification of the MMU.

```
assign inp =({1'b1,vsid[0:23],ea[4:9],ea[10:13]});
assign out0=({vld0,e0[0:23],e0[24:29],e0[54:57]});
assign out1=({vld1,e1[0:23],e1[24:29],e1[54:57]});
assign hit0=(inp == out0);
assign hit1=(inp == out1);
assign miss=~(hit0 | hit1);
```

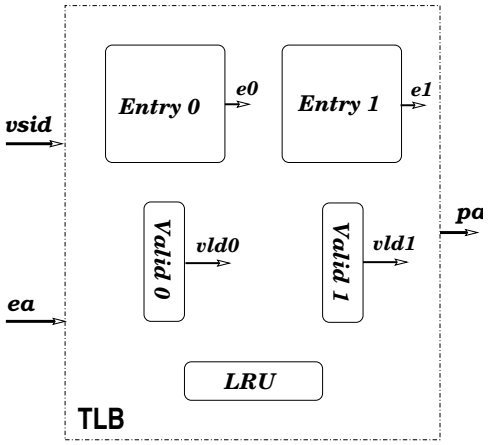


Figure 5.4: TLB block diagram

This property verifies miss detection for a two-way set-associative TLB. It would be a simple extension for generating this property for a *n*-way set-associative TLB. Here *vsid* (virtual segment id) and *ea* (effective address) are inputs and *pa* (physical address) is the output of the TLB block. The *e* and *vld* variables are outputs from the *entry* and *valid* blocks respectively as shown in Figure 5.4.

Similarly we have generated and validated the property for the BAT array miss detection. There are several mismatches found (between the reference model and the implementation) during property checking. The architecture specification document does not provide the value for the else condition (default value of a signal for example) in most of the cases. As a result the description of the property does not have the default value for a signal, whereas the signal has a definite value in its implementation under all possible conditions. Symbolic simulation produced mismatches in those cases. Consider the following read implementation of a SRAM cell.

```
assign out = (rdClk & rdEn) ? ram[rdAddr] : 32'b0;
```

This implementation assigns *32'b0* to signal *out* when condition (*rdClk & rdEn*) is *false*. However, the architecture document does not specify the value in the default case. As a result, the generated property does not have this value that caused the mismatch. The architecture document can be updated to add the values in all cases. It is also possible to impose certain constraints in Versys2 [41] to avoid the detection

of such false negatives. For example, we can set the condition (*rdClk* & *rdEn*) as *true* in the Versys2 configuration file to avoid the detection of the mismatch mentioned above.

5.2.2 Equivalence Checking of the DLX Architecture

We validated the DLX [22] processor using equivalence checking. We have chosen the DLX processor since it has been well studied in academia, and there are HDL implementations available that can be used in our validation framework. We obtained a VHDL description of the synthesizable 32-bit RISC DLX from *eda.org* [86] and used it as the *implementation*. The structure and behavior of the DLX architecture is captured using the EXPRESSION ADL. Our framework generated the VHDL description from the ADL specification using the method described in Section 4.3. The generated VHDL description is used as the *reference model* (specification) for the validation.

Regardless of the implementation style, the equivalence checker can verify a design based on the correct behavior in the reference model. For example, our HDL generation framework generates a 32-bit adder module that uses a carry-look-ahead principle. The equivalence checker verifies that this design is equivalent to the 32-bit adder implementation, which uses a ripple-carry adder principle. Equivalence checking took 4 seconds to verify the adder on a 300 MHz Sun Ultra-250 with 1024M RAM. Similarly, we generated a structural model of a 32×32 register file and used it as a reference model to verify the behavioral register file implementation [86]. In this case, equivalence checking took 432 seconds. The majority of this time (347 seconds) was consumed in the elaboration (linking) phase of the behavioral implementation.

Our framework generated synthesizable RTL for 32-bit RISC DLX that supports signed operations. To avoid memory explosion, we guided the RTL generation process to have a structure similar to the implementation [86]. The equivalence checking process took 397 seconds. We have encountered a mismatch in the output data bus at clock cycle 2500. The analysis revealed that the problem is in the overflow bit of the adder. The ripple-carry adder implementation of the DLX [86] had an incorrect

computation of the overflow bit.

Design analysis in our framework is easy once we figure out the module that is causing the problem. For example, in this particular case once we know that the adder is causing the problem, we can verify the adder implementation of the DLX by generating an adder specification (HDL description) from our framework and applying equivalence checking.

Table 5.1: Validation of the DLX implementation using equivalence checking

Reference Implementation	32-bit CLA adder ripple-carry adder	32×32 register-file behavioral model	32-bit DLX DLX [86]
Validation Time	4 seconds	432 seconds	397 seconds

Table 5.1 summarizes the experimental results. Each column in the table presents the equivalence checking time for the respective reference model and the implementation. As we can see from the table that the validation time is longer for equivalence checking of the register file than the DLX processor. This is due to the fact that the models used for verifying the register-file are very different (structural vs. behavioral). However, we have guided the reference model generation process of the DLX processor such that the reference model has structure similar to that of the implementation.

5.3 Chapter Summary

Verification is one of the most complex and expensive tasks in the current microprocessor design flow. A significant bottleneck in the validation of such systems is the lack of a golden reference model. Thus, many existing approaches employ bottom-up validation methodology by using a combination of simulation techniques and formal methods.

This chapter presented a top-down validation methodology driven by an ADL. The reference model (HDL description) is generated from the ADL specification of the architecture. An important aspect of our methodology is the ability to perform both model (property) checking and equivalence checking depending on the generated

reference model. Our framework generates the intended properties to enable model checking, and generates the RTL description of the processor to enable equivalence checking. To verify the properties, the framework uses Versys2 [41] that generates assertions from the reference model and applies them to the implementation using symbolic trajectory evaluation. The Formality [85] is used to perform equivalence checking. We have applied our methodology in two validation scenarios: property checking of a memory management unit of a microprocessor that is compliant with the PowerPC instruction-set, and equivalence checking of the DLX architecture.

Specification-driven hardware generation and validation of design implementation using equivalence checking has one limitation: the structure of the generated hardware model (reference) needs to be similar to that of the implementation. This requirement is primarily due the limitation of the equivalence checkers available today. Equivalence checking is not possible using these tools if the reference and implementation designs are large and drastically different. Property checking can be useful in such scenarios to ensure that both designs satisfy a set of properties. However, property checking does not guarantee equivalence between two designs. As a result, it is also necessary to use other complementary validation techniques (such as simulation) to verify the implementation.

Chapter 6

Functional Test Generation

As embedded systems continue to face increasingly higher performance requirements, deeply pipelined processor architectures are being employed to meet desired system performance. Functional validation of such programmable processors is widely acknowledged as a major bottleneck in current 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.

This chapter presents two specification-driven test generation techniques. Section 6.1 describes a model checking based functional test program generation technique for pipelined processors. Section 6.2 proposes a functional fault model that is used to define functional coverage for pipelined architectures. It also presents procedures for generating test programs to detect all the faults in the functional fault model. Finally, Section 6.3 summarizes the chapter.

6.1 Test Generation using Model Checking

This section presents a specification-driven test generation technique for pipelined processors. To make ADL-driven test generation applicable to realistic embedded

processors, three important 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 generate test programs. Finally, an efficient test generation technique is needed that can model complex designs and enable fast generation of functional test programs.

6.1.1 Test Generation Methodology

Figure 6.1 shows our graph based functional test program generation methodology. The processor architecture is specified in an ADL. The graph model of the processor is generated from the ADL specification. The properties are generated based on the graph coverage metric discussed later in this section. The properties are applied at the module level using the SMV model checker [28]. 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. This technique reduces the time and space required for generating test programs by applying properties at the module level and composing the responses in sequence by traversing the pipeline graph.

Algorithm 7 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 the primary inputs of the processor, the output requirements for the parent node of n are 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. The primary input assignments are converted into test programs (instruction sequences) by putting random values in the un-assigned inputs. The complexity of the algorithm is $O(n \times p)$, where n is the number of nodes in the graph model and p is the number of properties.

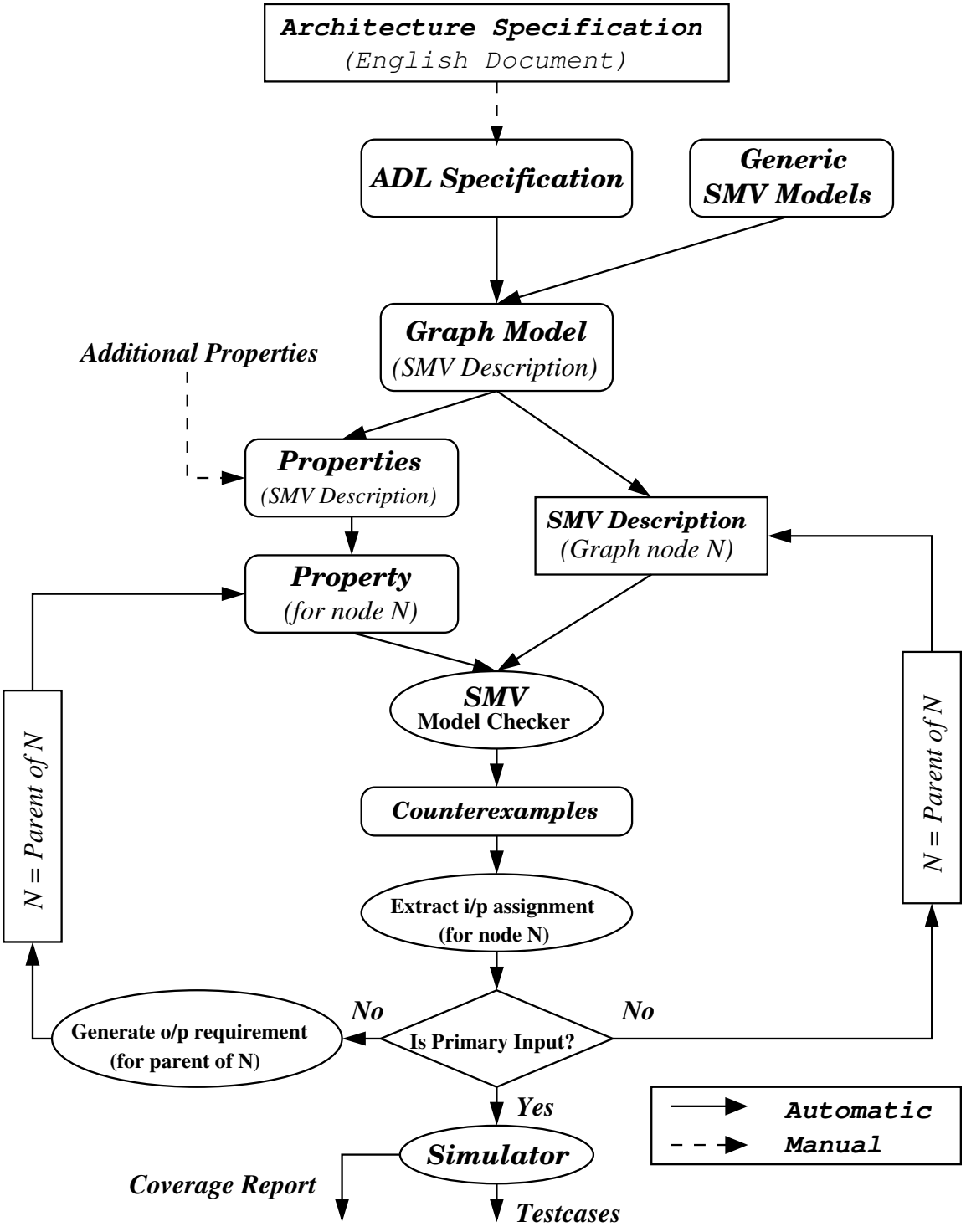


Figure 6.1: Test program generation methodology

Algorithm 7: *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
    $inputs = \phi$
    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

Graph Coverage

Measuring progress is an important task that enables the designer to decide when to end the verification effort. We propose a coverage metric based on functional coverage of the pipeline. We define all possible interactions between operations (instructions) and pipeline stages (paths) through graph coverage.

We define graph coverage using 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 the *flushed* state if an instruction in the node is flushed due to the occurrence of an exception in any of its successor 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 a parent node to a child node. An edge is *flushed* if the parent node is flushed due to the exception in the child node. The edge coverage conditions are redundant if a node has only one child. However, if a node has multiple children (or parents), edge coverage conditions are necessary.

Our test generation algorithm traverses the pipeline graph and generates properties based on the graph coverage described above. For example, consider the test generation for a feedback path (edge) from $MUL7$ to $IALU$ for the DLX architecture shown in Figure 3.6. 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.

6.1.2 A Case Study

In a case study we successfully applied the proposed methodology to the DLX processor [22]. Figure 3.6 shows the graph model of the DLX processor. The DLX architecture has five pipeline stages: fetch, decode, execute, memory (MEM), and writeback. 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.

Test Generation Results

This section describes the number of test cases generated for the DLX processor using the graph coverage described in Section 6.1.1. The DLX processor shown in

Figure 3.6 has 20 nodes and 24 edges (except feedback paths). We have described all the 91 instructions of the DLX processor [22].

Table 6.1: 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 6.1 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 use 91 test cases in this category to 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 6.2: 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 exercise 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 the 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 6.2 shows the number of reduced test programs in different categories.

† Same testcases as in the node coverage.

Test Program Generation: An Example

Example 6.1: *Consider a fragment of the DLX pipeline containing three internal registers of the division unit (DIV) as shown in Figure 6.2. The goal is to initialize two registers A_{in} and B_{in} with values 2 and 3 respectively at clock cycle 9.*

In this section we describe our test generation approach using Example 6.1. 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., $result = C_{out}$.

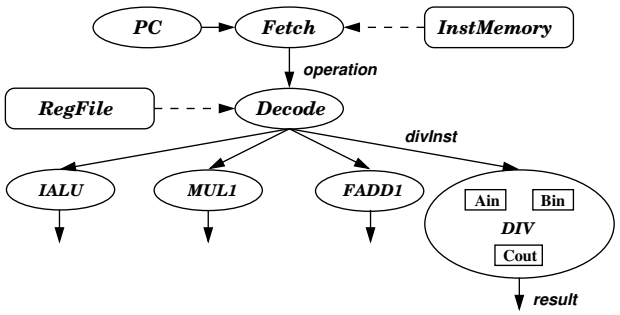


Figure 6.2: A fragment of the DLX architecture

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 the SMV language [28]. 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 to the complete description of the processor, SMV takes 375.98 seconds on a 333 MHz Sun UltraSPARC-II with 128M RAM, and requires 1928568 BDD nodes to generate the counterexample. In the remainder of this section, we illustrate how our test generation methodology improves both time and space requirements for the Example 6.1.

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 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.

```
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 of 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
```

For this example, 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.

If the property is applied to the complete description of the processor, SMV takes 375.98 seconds and requires 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.

6.2 Functional Coverage driven Test Generation

Several coverage measures are commonly used during design validation, such as code coverage, toggle coverage and fault coverage. Unfortunately, these measures do not have any direct relationship to the functionality of the device. For example, none of these determine if all possible interactions of hazards, stalls and exceptions are tested in a processor pipeline. There is a need for a coverage metric based on the functionality of the design. To define a useful functional coverage metric, we need to define a fault model of the design that is described at the functional level and independent of the implementation details.

In this section, we present a functional fault model for pipelined processors. The

fault model should be applicable to the wide varieties of today's microprocessors from various architectural domains such as RISC, DSP, VLIW, and superscalar. These architectures differ widely in terms of their structure (organization) and behavior (instruction-set). We have developed a graph-theoretic model that can capture a wide spectrum of pipelined processors, coprocessors, and heterogeneous memory subsystems. We have defined functional coverage based on the effects of faults in the fault model applied at the level of the graph-theoretic model. This allows us to compute functional coverage of a pipelined processor for a given set of random or constrained-random test sequences. We present test generation procedures that accept the graph model of the pipelined processor as input and generate test programs to detect all the faults in the functional fault model.

6.2.1 Functional Fault Models

The universe of design errors consists of many types of faults including functional (logical) faults that affect the logic function, and timing faults that effect the operating speed of the system. We only consider the functional faults. The set of possible functional faults (bugs) is dependent on the functionality of the design. In this section, we present fault models for various functions in a pipelined processor. We categorize various computations in a pipelined processor into *register read/write*, *operation execution*, *execution path* and *pipeline execution*. We outline the underlying fault mechanisms for each fault model, and describe the effects of these faults at the level of the graph-based architecture model presented in Section 3.1.1.

Fault Model for Register Read/Write

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

Fault Model for Operation Execution

All operations must execute correctly if there are no faults. In the presence of a fault, the output of the computation is different from the expected output. The fault could be due to an error in operation decoding, control generation or final computation. Erroneous operation decoding might return an incorrect opcode. This can happen if incorrect bits are decoded for 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 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 content of the destination is not equal to val_i.

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[1]. We define all these activated paths as the *execution path* for that operation. An execution path ep_{op_i} is faulty if it produces an 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 is 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

[1]pipeline and data-transfer paths are described in Section 3.1.1

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 an incorrect value, i.e., $\exists k\ dest_k \neq val_i^k$.

Fault Model for Pipeline Execution

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

Let us define the stall set for a unit u (say 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 say) as all possible ways to create an exception in that unit. We define the set of all possible multiple exception scenarios as $MESS$. Hence, the exception set for the architecture $ExceptionSet = \cup_{\forall u} ES_u \cup MESS$. We consider two types of pipeline interactions: stalls and exceptions. Therefore, all possible pipeline interactions (PIs) can be defined as: $PIs = StallSet \cup ExceptionSet$. Let us assume a sequence of operations ops_{pi} causes a pipeline interaction pi (i.e., $pi \in PIs$), and updates n storage locations.

Let val_{pi} denote the result of computing the operation sequence ops_{pi}. The val_{pi} has n components ($\cup_{k=1}^{n} val_{pi}^k$). In the fault-free case, the destinations will contain correct values, i.e., $\forall k\ dest_k = val_i^k$. Under a fault, at least one of the destinations will have an incorrect value, i.e., $\exists k\ dest_k \neq val_i^k$

6.2.2 Functional Coverage Estimation

We define functional coverage based on the fault models described in Section 6.2.1.

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

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

6.2.3 Test Generation Techniques

In this section, we present test generation procedures for detecting faults covered by the fault models presented in Section 6.2.1. Different architectures have specific instructions to observe the contents of registers and memories. In our framework, 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* (Procedure 1) that is used by the test generation algorithms. Procedure 1 accepts a list of operations as input and returns the modified list of operations. First, 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* ∪ *initOper*;
            endif
        endfor
        resOperations = *resOperations* ∪ *oper*;
        readOper: create an operation to read the destination of *oper*;
        resOperations = *resOperations* ∪ *readOper*;
    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*. Procedure 1 assigns a register *R3* to *dest* field and an immediate value to *src2* field. It also creates an initialization operation for the source *R1*. Finally, it creates an operation to read the destination. The modified list consists of three operations (in that order): *MOVI R1 0×5*, *ADD R3 R5 0×23*, and *STORE R3 R6 0×0*.

Test Generation for Register Read/Write

Algorithm 8 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 6.2.1. 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, 0×0". The move-immediate (MOVI) instruction writes val_i in register R_i. The STORE instruction reads the content of R_i and writes it in memory addressed by R_j (offset 0).

Algorithm 8: *Test Generation for Register Read/Write*
Input: Graph model of the architecture *G*.
Output: Test programs for detecting faults in register read/write.
begin /*** TestProgramList = {} ***/
  for each register *reg* in architecture *G*
    $value_{reg}$ = GenerateUniqueValue(*reg*);
    writeInst = an instruction that writes $value_{reg}$ in register *reg*.
    $testprog_{reg}$ = createTestProgram(*writeInst*)
    TestProgramList = *TestProgramList* ∪ $testprog_{reg}$;
  endfor
  return *TestProgramList*.
end

Theorem 6.2.1 *The test sequence generated using Algorithm 8 is capable of detecting any detectable fault in the register read/write fault model.*

Proof Algorithm 8 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 than the written value. ■

Test Generation for Operation Execution

Algorithm 9 presents the procedure for generating test programs for detecting faults in operation execution. The fault model for the operation execution is described in Section 6.2.1. 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 3.3 has

four operations: two initialization operations ("MOV R3 0×333", "MOV R5 0×212") followed by the ADD operation ("ADD R2 R3 R5"), followed by the reading of the result ("STORE R2, Rx, 0×0").

Algorithm 9: *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

Theorem 6.2.2 *The test sequence generated using Algorithm 9 is capable of detecting any detectable fault in the operation execution fault model.*

Proof Algorithm 9 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. ■

Test Generation for Execution Path

Algorithm 10 presents the procedure for generating test programs for detecting faults in execution path. The fault model for the execution path is described in Section 6.2.1. 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 activate different set of edges in the execution path, it generates all possible source/destination assignments for each operation in the operation group.

Algorithm 10: *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

Theorem 6.2.3 *The test sequence generated using Algorithm 10 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 a 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 10 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.

Test Generation for Pipeline Execution

Algorithm 11 presents the procedure for generating test programs for detecting faults in pipeline execution. The fault model for the pipeline execution is described in Section 6.2.1. 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 generation of multiple exceptions during execution of the combined test program.

Theorem 6.2.4 *The test sequence generated using Algorithm 11 is capable of detecting any detectable fault in the pipeline execution.*

Proof Algorithm 11 generates test programs for all possible interactions during pipeline execution. The first for loop (L1) generates all possible hazard and exception scenarions 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. ■

6.2.4 A Case Study

We have applied our methodology on two pipelined architectures: a VLIW implementation of the DLX architecture [22], and a RISC implementation of the SPARC V8 architecture [32].

[2]The unit closer to completion has higher order.

Algorithm 11: *Test Generation for Pipeline Execution*

Input: Graph model of the architecture G.

Output: Test programs for detecting faults in pipeline execution.

begin /*** TestProgramList = {} ***/

L1: **for** each unit node *unit* in architecture G

L2: **for** each exception *exon* possible in *unit*

$template_{exon}$ = template for exception *exon*

$testprog_{unit}$ = createTestProgram($template_{exon}$);

$TestProgramList = TestProgramList \cup testprog_{unit}$;

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

Experimental Setup

We have developed our test generation and coverage analysis framework using Verisity's Specman Elite [93]. We have captured executable specification of the architectures using Verisity's "e" language. This includes description of 91 instructions for the DLX, and 106 instructions for the SPARC V8 architecture. We refer these as *specifications*. We have implemented a VLIW version of the DLX architecture (shown in Figure 3.6) using Verisity's "e" language. It contains five pipeline stages: fetch, decode, execute, memory and writeback. The execute stage has four parallel execution paths: an ALU, a four-stage floating-point adder, a seven-stage multiplier, and a multi-cycle divider. We have used the LEON2 processor [43] that is a VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. We refer these models (VLIW DLX and LEON2) as *implementations*.

Our framework generates test programs in three different ways: random, constrained random, and our approach. Specman Elite [93] 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, we have used the highest probability for choosing register-type operations in DLX to generate test programs for register read/write. Since, register-type operations have three register operands, the chances of reading/writing registers are higher than immediate type (two register operands) or branch type (one register operand) operations. The test programs generated by our approach uses the algorithms described in Section 6.2.3. To ensure that the generated test programs are executed correctly, our framework applies the test programs on the implementation as well as the specification, and compares the contents of the program counter, registers and memory locations after execution of each test program.

The Specman Elite framework allows definition of various coverage measures that enables us to compute the functional coverage described in Section 6.2.2. 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 have 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.

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 6.3: Test programs for validation of DLX architecture

Fault Models	Test Generation Techniques		
	Random	Constrained	Our Approach
Register Read/Write	3900 (100%)	750 (100%)	128 (100%)
Operation Execution	437 (100%)	443 (100%)	91 (100%)
Execution Path	12627 (100%)	1126 (100%)	160 (100%)
Pipeline Execution	30000 (25%)	30000 (30%)	322 (100%)

Table 6.3 shows the comparative results for the DLX architecture. The rows indicate the fault models, and the columns indicate test generation techniques. An entry in the table has two numbers. The first one represents the number of operations 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. The number 100% implies that the generated test programs covered all the faults in that fault model. For example, the *Random* technique covered all the faults in "*Register Read/Write*" function using 3900 tests. The number of test programs for operation execution are similar for both random

and constrained-random approaches. This is because the constraint used in this case (same probability for all opcodes) may be the default option used in random test generation approach.

Table 6.4: Test programs for validation of LEON2 processor

Fault Models	Test Generation Techniques		
	Random	Constrained	Our Approach
Register Read/Write	1746 (100%)	654 (100%)	128 (100%)
Operation Execution	416 (100%)	467 (100%)	106 (100%)
Execution Path	1500 (100%)	475 (100%)	96 (100%)
Pipeline Execution	30000 (48%)	30000 (50%)	56 (100%)

Table 6.4 shows the comparative results for different test generation approaches for the LEON2 processor. The trend is similar in terms of number of operations and functional coverage for both the DLX and LEON2 architectures. The random and constrained-random approaches have obtained 100% functional coverage for the first three fault models using an order of magnitude more test vectors than our approach. We have analyzed the cause for the low functional 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 the bigger pipeline structure (larger set of pipeline interactions) in the VLIW DLX, it has lower fault coverage than the LEON2 architecture in *pipeline execution*.

6.3 Chapter Summary

Specification-driven test program generation is a promising approach for functional validation of pipelined processors. In this chapter, we presented two test generation techniques. The first half of the chapter presented a model checking based functional test program generation technique for pipelined processors. Our methodology accepts an ADL specification of the processor as input. A graph model of the pipelined processor is generated from the ADL specification. We defined the functional coverage of the pipeline behavior in terms of the graph coverage. We presented

a test program generation algorithm that traverses the pipeline graph to generate test programs based on the coverage metric. Our technique reduced the test generation time and the required BDD size by an order of magnitude.

The second half of the chapter presented a functional coverage based test generation technique for pipelined architectures. The methodology made two important contributions. First, we presented a functional fault model that is used in defining the functional coverage. Second, we presented test generation procedures that accept the graph model of the microprocessor as input and generate test programs to detect all the faults in the functional fault model. We are able to measure the goodness of a given set of random test sequences using our functional coverage metric. We applied this technique on two pipelined architectures: DLX and LEON2. Our experimental results demonstrate that the required number of test sequences generated by our algorithms to obtain a given fault (functional) coverage is an order of magnitude less than the random or constrained-random test programs.

Chapter 7

Conclusions and Future Work

There is no argument that validation is one of the most important problems in today's SOC design methodology. A significant bottleneck in the validation of programmable embedded systems is the lack of a golden reference model. As a result, many existing approaches employ a bottom-up validation approach by using a combination of simulation techniques and formal methods. This thesis presented a top-down validation methodology for programmable architectures that complements the existing bottom-up techniques. This chapter draws the conclusions from the research results obtained, and looks at some future work on specification-driven validation and related issues.

7.1 Conclusions

This thesis investigates several issues related to top-down validation of programmable embedded systems consisting of processor core, coprocessors, and memory subsystem. There are four important problems to be addressed in a specification-driven validation methodology:

- **Specification**: How to capture a wide variety of programmable architectures using a specification language? The language should be powerful enough to specify the wide spectrum of contemporary processor, coprocessor, and memory

features. On the other hand, the language should be simple enough to allow correlation of the information between the specification and the architecture manual.

- ❒ **Specification Validation**: How to validate the architecture specification to ensure it is golden? Specification analysis and validation would be an easier task if the specification language has formal semantics.

- ❒ **Model Generation**: How to generate hardware, simulation models, and models for other validation techniques from the given specification?

- ❒ **Design Validation**: What are the bottom-up validation techniques that the top-down methodology can complement?

This thesis examines all of the problems mentioned above. We used the EXPRESSION ADL [20] to specify the architecture. It can capture the structure and behavior of a wide variety of programmable architectures including RISC, DSP, VLIW, and superscalar. The validation techniques, we developed, are applicable to any specification language that captures both the structure and the behavior of the architecture.

We developed validation techniques to ensure that the static behavior of the pipeline is well-formed by analyzing the structural aspects of the specification using a graph based model. The dynamic behavior is verified by analyzing the instruction flow in the pipeline using a FSM-based model to validate several architectural properties such as determinism and in-order execution in the presence of hazards and multiple exceptions. These properties are by no means complete to prove the correctness of the specification. The designer can add new architecture specific properties and easily integrate it in our validation framework.

A major challenge in top-down validation methodology is the ability to generate executable models from the specification for a wide variety of programmable architectures. We defined a functional abstraction technique to enable generation of models for simulation, hardware generation, and property checking from the ADL specification. The generated simulation and hardware models are used for design space exploration of programmable architectures.

This thesis explored two top-down validation scenarios that complement existing bottom-up techniques: design validation and test generation. The generated hardware is used as a reference model for verifying the hand-written RTL implementation using a combination of symbolic simulation and equivalence checking. We also developed specification-driven test generation techniques based on the functional coverage of the pipelined architectures.

7.2 Future Research Directions

Top-down validation of programmable embedded systems is a major problem. We believe that we explored only the tip of the iceberg. There are many challenges remaining to make this approach viable in practice. The work presented in this thesis can be extended in the following directions:

- ADLs allow ease of specification for programmable architectures. Formal languages allow specification in a rigorous form. An interesting direction is to develop a specification language that combines the benefits of both. Such a language would make specification validation a easier problem.

- There are two important problems that needs to be investigated during specification validation. First, it is necessary to develop architecture specific properties such as validation of execution style for an out-of-order superscalar processor. Second, it is important to develop a completeness criteria (to establish both necessary and sufficient conditions) for specification validation.

- The functional abstraction based approach we developed in this thesis allows model generation for uni-processor architectures. There is a need for a methodology to generate models from the specification of programmable architectures containing multiple processor cores.

- Further studies can be done in the design space exploration of architectures. We have done processor-memory exploration using small and medium sized benchmarks and observed many interesting trade-offs. Further experiments can

be performed using larger applications, to study the impact of different parts of the application (such as loop nests) on the memory organization behavior and overall performance, as well as on system power.

➤ Our synthesis-driven exploration framework can be used for generating hardware models for real-world architectures. We have not considered the optimization and resource sharing issues of our data path components yet. As a result, the execution units consumes 60-70% of the total area and power of the generated hardware model. Future research can enable generation of optimized data path components with shared resources.

➤ Specification-driven hardware generation and validation of design implementation using generated hardware model has one limitation: the generated hardware model (reference) should have a structure similar to the implementation. The requirement is primarily due to the limitation of the equivalence checkers available today. It is an easier task for the tool when the module boundaries are same. Equivalence checking is not possible using these tools if the designs are large (in the order of a million gates) and the reference and implementation designs are drastically different. In reality, the implementation goes through numerous optimizations due to various requirements such as cost, area, power and performance. As a result, the final implementation might not have similar structure as intended in the original specification. There is a need for a new validation technique that would enable reference model generation and design validation without any knowledge of the implementation details.

➤ The generated test programs are applied to a cycle-accurate simulator. It would be interesting to perform functional validation of RTL implementation using the generated test programs. We have investigated the applicability of our technique on two simple pipelined processors: DLX and LEON2. Applicability of these techniques can be investigated on today's microprocessors. It is necessary to perform further comparative studies with our functional coverage metric against existing coverage measures, such as code coverage and stuck-at coverage.

➤ This thesis considered programmable embedded systems consisting of processor core, coprocessor, and memory subsystem. Traditional embedded systems contain many more components including DMAs, input/output devices, specific hardwares, buses, and so on. It is necessary to extend the current methodology for specification, model generation, and top-down validation of embedded systems.

Bibliography

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

[2] H. Akaboshi. *A Study on Design Support for Computer Architecture Design.* PhD thesis, Dept. of Information Systems, Kyushu University, Japan, Jan 1996.

[3] H. Akaboshi and H. Yasuura. Behavior extraction of MPU from HDL description. In *Proceedings of Asia Pacific Conference on Hardware Description Languages (APCHDL)*, 1994.

[4] D. Anastasakis, R. Damiano, H. Ma, and T. Stanion. A practical and efficient method for compare-point matching. In *Proceedings of Design Automation Conference (DAC)*, pages 305–310, 2002.

[5] ARC. *http://www.arccores.com.* ARC Cores.

[6] Axys. *Axys Design Automation.* http://www.axysdesign.com.

[7] Chris Basoglu, Woobin Lee, and John Setel O'Donnell. *The MAP1000A VLIW Mediaprocessor*, 2000.

[8] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 83–94, 2000.

[9] R. Bryant. Symbolic simulation - techniques and applications. In *Proceedings of Design Automation Conference (DAC)*, pages 517–521, 1990.

[10] R. Bryant and C. Seger. Formal verification of digital circuits using symbolic ternary system models. In *Proceedings of Computer Aided Verification (CAV)*, pages 121–146, 1990.

[11] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In D. Dill, editor, *Proceedings of Computer Aided Verification (CAV)*, volume 818 of *LNCS*, pages 68–80. Springer-Verlag, 1994.

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

[13] H. Chockler, O. Kupferman, R. Kurshan, and M. Vardi. A practical approach to coverage in model checking. In *Proceedings of Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 66–78. Springer-Verlag, 2001.

[14] Paul C. Clements. A survey of architecture description languages. In *Proceedings of International Workshop on Software Specification and Design (IWSSD)*, pages 16–25, 1996.

[15] D. Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical report, SRI-CSL-93-12, 1993.

[16] C. Ejik. Sequential equivalence checking without state space traversal. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 618–623, 1998.

[17] M. Freericks. The nML machine description formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Dept., 1993.

[18] P. Grun, A. Halambi, N. Dutt, and A. Nicolau. RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions. In *Proceedings of International Symposium on System Synthesis (ISSS)*, pages 44–50, 1999.

[19] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *Proceedings of Design Automation Conference (DAC)*, pages 299–302, 1997.

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

[21] A. Halambi, A. Shrivastava, N. Dutt, and A. Nicolau. A customizable compiler framework for embedded systems. In *Proceedings of Software and Compilers for Embedded Systems (SCOPES)*, 2001.

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

[23] P. Ho, A. Isles, and T. Kam. Formal verification of pipeline control using controlled token nets and abstract interpretation. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 529–536, 1998.

[24] R. M. Hosabettu. *Systematic Verification Of Pipelined Microprocessors.* PhD thesis, Department of Computer Science, University of Utah, 2000.

[25] http://www-cad.eecs.berkeley.edu/Software/software.html. *Espresso.*

[26] http://www-ee.engr.ccny.cuny.edu/notes/ee210/eqntott_man.html. *Eqntott.*

[27] http://www.coware.com. *CoWare LISATek Products.*

[28] http://www.cs.cmu.edu/~modelcheck. *Symbolic Model Verifier.*

[29] http://www.lucent.com/micro/Starcore. *Starcore, Next Generation DSPs.*

[30] http://www.motorola.com. *MPC7450 Microprocessor.*

[31] http://www.sgi.com. *MIPS R10000 Microprocessor.*

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

[33] http://www.ti.com/sc/docs/products/dsp/C6000/index.htm. *$TMS320C6000^{TM}$ DSP*.

[34] A. Inoue, H. Tomiyama, F. Eko, H. Kanbara, and H. Yasuura. A programming language for processor based embedded systems. In *Proceedings of Asia Pacific Conference on Hardware Description Languages (APCHDL)*, pages 89–94, 1998.

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

[36] J. Paakki. Attribute grammar paradigms - a high level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–256, June 1995.

[37] R. Jhala and K. L. McMillan. Microarchitecture verification by compositional model checking. In G. Berry et al., editor, *Proceedings of Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 396–410. Springer-Verlag, 2001.

[38] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 364–373, 1990.

[39] A. Kejariwal, P. Mishra, J. Astrom, and N. Dutt. HDLGen: Automatic HDL generation driven by an architecture description language. Technical Report CECS 03-04, University of California, Irvine, 2003.

[40] A. Khare, N. Savoiu, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. V-SAT: A visual specification and analysis tool for system-on-chip exploration. In *Proceedings of EUROMICRO Conference*, pages 1196–1203, 1999.

[41] N. Krishnamurthy, M. Abadir, A. Martin, and J. Abraham. Design and development paradigm for industrial formal verification tools. *IEEE Design & Test of Computers*, 18(4):26–35, July-August 2001.

[42] D. Lanneer, J. Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. CHESS: Retargetable code generation for embedded DSP processors. In *Code Generation for Embedded Processors.*, pages 85–102. Kluwer Academic Publishers, 1995.

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

[44] R. Leupers and P. Marwedel. Retargetable generation of code selectors from HDL processor models. In *Proceedings of European Design and Test Conference (EDTC)*, pages 140–144, 1997.

[45] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1):75–108, 1998.

[46] J. Marques-Silva and T. Glass. Combinational equivalence checking using satisfiability and recursive learning. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 145–149, 1999.

[47] N. Medvidovic and R. Taylor. A framework for classifying and comparing architecture description languages. In M. Jazayeri and H. Schauer, editors, *Proceedings of European Software Engineering Conference (ESEC)*, pages 60–76. Springer–Verlag, 1997.

[48] MIPS Technologies, Inc. *MIPS R4000 Microprocessor User's Manual*, 1994.

[49] P. Mishra, J. Astrom, N. Dutt, and A. Nicolau. Functional abstraction of programmable embedded systems. Technical Report UCI-ICS 01-04, University of California, Irvine, January 2001.

[50] P. Mishra and N. Dutt. Automatic functional test program generation for pipelined processors using model checking. In *Proceedings of High Level Design Validation and Test (HLDVT)*, pages 99–103, 2002.

[51] P. Mishra and N. Dutt. Modeling and verification of pipelined embedded processors in the presence of hazards and exceptions. In *Proceedings of Distributed and Parallel Embedded Systems (DIPES)*, pages 81–90, 2002.

[52] P. Mishra and N. Dutt. Automatic modeling and validation of pipeline specifications. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(1), 2004.

[53] P. Mishra and N. Dutt. Functional coverage driven test generation for validation of pipelined processors. Technical Report CECS 04-05, University of California, Irvine, 2004.

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

[55] P. Mishra, N. Dutt, N. Krishnamurthy, and M. Abadir. A top-down methodology for validation of microprocessors. *IEEE Design & Test of Computers*, 2004.

[56] P. Mishra, N. Dutt, and A. Nicolau. Automatic validation of pipeline specifications. In *Proceedings of High Level Design Validation and Test (HLDVT)*, pages 9–13, 2001.

[57] P. Mishra, N. Dutt, and A. Nicolau. Functional abstraction driven design space exploration of heterogeneous programmable architectures. In *Proceedings of International Symposium on System Synthesis (ISSS)*, pages 256–261, 2001.

[58] P. Mishra, N. Dutt, and A. Nicolau. Specification of hazards, stalls, interrupts, and exceptions in EXPRESSION. Technical Report UCI-ICS 01-05, University of California, Irvine, 2001.

[59] P. Mishra, N. Dutt, and A. Nicolau. A study of out-of-order completion for the MIPS R10K superscalar processor. Technical Report UCI-ICS 01-06, University of California, Irvine, January 2001.

[60] P. Mishra, N. Dutt, and H. Tomiyama. Towards automatic validation of dynamic behavior in pipelined processor specifications. *Kluwer Design Automation for Embedded Systems*, 8(2-3):249–265, June-September 2003.

[61] P. Mishra, P. Grun, N. Dutt, and A. Nicolau. Processor-memory co-exploration driven by an architectural description language. In *Proceedings of International Conference on VLSI Design*, pages 70–75, 2001.

[62] P. Mishra, A. Kejariwal, and N. Dutt. Rapid exploration of pipelined processors through automatic generation of synthesizable RTL models. In *Proceedings of Rapid System Prototyping (RSP)*, pages 226–232, 2003.

[63] P. Mishra, A. Kejariwal, and N. Dutt. Synthesis-driven exploration of pipelined embedded processors. In *Proceedings of International Conference on VLSI Design*, 2004.

[64] P. Mishra, M. Mamidipaka, and N. Dutt. A framework for memory subsystem exploration. Technical Report CECS 02-19, University of California, Irvine, 2002.

[65] P. Mishra, M. Mamidipaka, and N. Dutt. Processor-memory co-exploration using an architecture description language. *To appear, ACM Transactions on Embedded Computing Systems (TECS)*, 3(1), 2004.

[66] P. Mishra, F. Rousseau, N. Dutt, and A. Nicolau. Architecture description language driven design space exploration in the presence of coprocessors. In *Proceedings of Synthesis and System Integration of Mixed Technologies (SASIMI)*, 2001.

[67] P. Mishra, F. Rousseau, N. Dutt, and A. Nicolau. Coprocessor codesign for programmable architectures. Technical Report UCI-ICS 01-13, University of California, Irvine, April 2001.

[68] P. Mishra, H. Tomiyama, N. Dutt, and A. Nicolau. Architecture description language driven verification of in-order execution in pipelined processors. Technical Report UCI-ICS 01-20, University of California, Irvine, 2000.

[69] P. Mishra, H. Tomiyama, N. Dutt, and A. Nicolau. Automatic verification of in-order execution in microprocessors with fragmented pipelines and multicycle functional units. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 36–43, 2002.

[70] P. Mishra, H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. Automatic modeling and validation of pipeline specifications driven by an architecture description language. In *Proceedings of Asia South Pacific Design Automation Conference (ASPDAC) / International Conference on VLSI Design*, pages 458–463, 2002.

[71] J. M. Mulder, N. T. Quach, and M. J. Flynn. An area model for on-chip memories and its application. *IEEE Journal of Solid State Circuits*, SC-26(1):98–105, Feb 1991.

[72] S. Pasricha, P. Biswas, P. Mishra, A. Shrivastava, A. Mandal, N. Dutt, and A. Nicolau. A framework for GUI-driven design space exploration of a MIPS4K-like processor. Technical Report CECS 03-17, University of California, Irvine, 2003.

[73] P. Paulin, C. Liem, T. May, and S. Sutarwala. FlexWare: A flexible firmware development environment for embedded systems. In *Prof. of Dagstuhl Workshop on Code Generation for Embedded Processors*, pages 67–84, 1994.

[74] V. Rajesh and Rajat Moona. Processor modeling for hardware software codesign. In *Proceedings of International Conference on VLSI Design*, pages 132–137, 1999.

[75] J. Sawada and W. D. Hunt. Processor verification with precise exceptions and speculative execution. In A. Hu and M. Vardi, editor, *Proceedings of Computer Aided Verification (CAV)*, volume 1427 of *LNCS*, pages 135–146. Springer-Verlag, 1998.

[76] C. Seger and R. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. In *Formal Methods in System Design*, volume 6, pages 147–189, March 1995.

[77] J. Shen, J. Abraham, D. Baker, T. Hurson, M. Kinkade, G. Gervasio, C. Chu, and G. Hu. Functional verification of the equator MAP1000 microprocessor. In *Proceedings of Design Automation Conference (DAC)*, pages 169–174, 1999.

[78] SIA. *National technology roadmap for semiconductors: Technology needs*. Semiconductor Industry Association, 1998.

[79] C. Siska. A processor description language supporting retargetable multi-pipeline DSP program development tools. In *Proceedings of International Symposium on System Synthesis (ISSS)*, pages 31–36, 1998.

[80] J. Skakkebaek, R. Jones, and D. Dill. Formal verification of out-of-order execution using incremental flushing. In A. Hu and M. Vardi, editor, *Proceedings of Computer Aided Verification (CAV)*, volume 1427 of *LNCS*, pages 98–109. Springer-Verlag, 1998.

[81] M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. In *IEEE Software*, volume 7(5), pages 52–64, 1990.

[82] StrongArm. *StrongARM Processors*. http://developer.intel.com, 2000.

[83] SUN Microsystems. *UltraSPARC IIi User's Manual*, 1997.

[84] Synopsys. *http://www.synopsys.com*.

[85] Synopsys Formality. *http://www.synopsys.com*.

[86] Synthesizable DLX. *http://www.eda.org/rassp/vhdl/models/processor.html*.

[87] Target. *http://www.retarget.com*. Target Compiler Technologies.

[88] Tensilica. *http://www.tensilica.com*. Tensilica Inc.

[89] Texas Instruments. *TMS320C6201 CPU and Instruction Set Reference Guide*, 1998.

[90] Trimaran. *The MDES User Manual*. http://www.trimaran.org, 1997.

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

[92] M. Velev and R. Bryant. Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction. In *Proceedings of Design Automation Conference (DAC)*, pages 112–117, 2000.

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

[94] L. Wang, M. Abadir, and N. Krishnamurthy. Automatic generation of assertions for formal verification of PowerPC microprocessor arrays using symbolic trajectory evaluation. In *Proceedings of Design Automation Conference (DAC)*, pages 534–537, 1998.

[95] www.intel.com. *IA-64 Architecture*.

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