Functional Validation of Programmable Architectures*

Prabhat Mishra Nikil Dutt

Architectures and Compilers for Embedded Systems (ACES) Laboratory
Center for Embedded Computer Systems, University of California, Irvine, USA
{pmishra, dutt}@ics.uci.edu

Abstract

Validation of programmable architectures, consisting of processor cores, coprocessors, and memory subsystems, is one of the major bottlenecks in current System-on-Chip design methodology. A critical challenge in validation of such systems is the lack of a golden reference model. Traditional validation techniques employ different reference models depending on the abstraction level and verification task (e.g., functional simulation or property checking), resulting in potential inconsistencies between multiple reference models. This paper presents a validation methodology that uses an Architecture Description Language (ADL) based specification as a golden reference model for validation of programmable architectures, and generation of executable models such as simulators and hardware prototypes. We present a validation framework that uses the generated hardware as a reference model to verify the hand-written implementation using a combination of symbolic simulation and equivalence checking. We also present functional coverage based test generation techniques for validation of pipelined processor architectures. Finally, the generated simulator and hardware models are also used for early exploration of programmable architectures.

1 Introduction

Computing is an integral part of daily life. We encounter two types of computing devices everyday: desktop based computing devices and embedded computer systems. Desktop based computing systems encompass traditional "computers", including personal computers (PC), notebook computers, workstations, and servers. Embedded computer systems are ubiquitous - they 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. Both types of computing devices use programmable components such as processors, coprocessors and memories to execute the application programs. In this paper, we refer these programmable components as *programmable architectures*.

The complexity of the programmable architectures is increasing at an exponential rate. There are two factors that contribute to this complexity growth: technology and demand. First, there is an exponential growth in the number of transistors per integrated circuit, as characterized by Moore's Law [6]. This trend has enabled an exponential increase in computational capacity, which fuels the second trend: the realization of ever more complex applications (e.g., in communication, multimedia, networking, and entertainment).

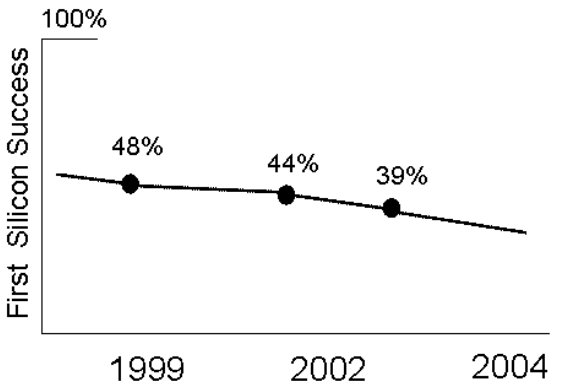

Figure 1. North America Re-spin Statistics

However, the complexity of designing and verifying such systems is also increasing at an exponential rate. Figure 1 shows a recent study on the number of first silicon re-spins of system-on-chip (SOC) designs in North America [31]. Almost half of the designs fail the very first time. This failure has tremendous impact on cost for two reasons. First, the delay in getting the working silicon drastically reduces the market share. Second, the fabrication cost is extremely high. The same study also concluded that *71% of SOC re-spins are due to logic bugs.*

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

Another study highlights the challenges for functional verification: Figure 2 shows the statistics of the SOC designs in terms of design complexity (logic gates), design time (engineer years), and verification complexity (simulation vectors) [31]. The study highlights the tremendous complexity faced by simulation-based validation of complex SOCs: it estimates that by 2007, a complex SOC will need 2000 engineer years to write 25 million lines of RTL code and one trillion simulation vectors for functional verification. A similar trend can be observed in the high-performance microprocessor space. Figure 3 summarizes a study of the pre-silicon bugs found in the Intel IA32 family of microarchitectures. This trend again shows an exponential increase in the number of logic bugs: a growth rate of 300-400% from one generation to the next. The bug rate is linearly proportional to the number of lines of structural RTL code in each design, indicating a roughly constant density.

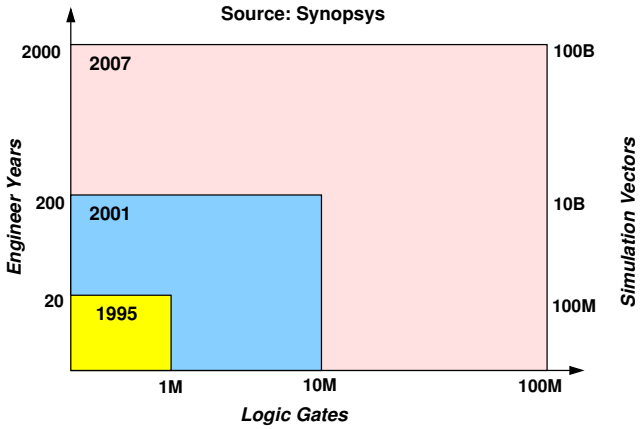

Figure 2. Complexity of SOC Verification

Simple extrapolation indicates that unless a radically new approach is employed, we can expect to see 20-30K bugs designed into the next generation and 100K in the subsequent generation. Clearly – in the face of shrinking time-to-markets – the amount of validation effort rapidly becomes intractable, and will significantly impact product schedules, with the additional risk of shipping products with undetected bugs.

The next obvious question is - where do all these bugs come from? An Intel report summarized the results of a statistical study of the 7855 bugs found in the Pentium 4 processor design prior to initial tapeout [2]. Figure 4 shows a breakdown of the results of the study.

Although "complexity" is ranked eighth on the list of bug causes, it is clear that it contributes to many of the categories listed above. More complex microarchitectures need more extensive documentation to describe them; they require larger design teams to implement them, increasing the likelihood of miscommunication between team members; and they introduce more corner cases, resulting in undiscovered bugs. Hence, microarchitectural complexity is the major contributor of the logic bugs.

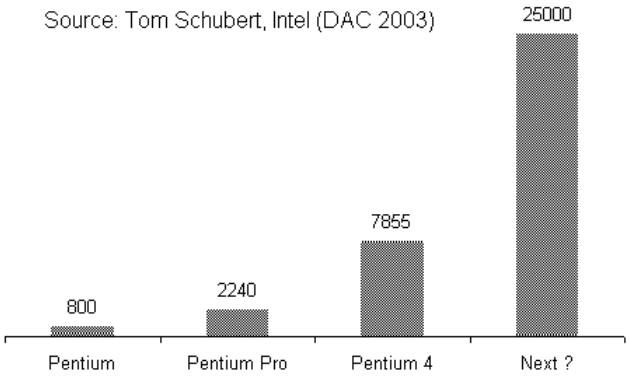

Figure 3. Pre-silicon Logic Bugs per Generation

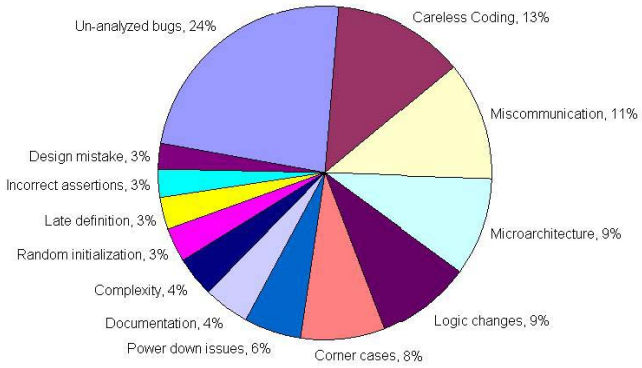

Figure 4. Pentium 4 Bug Breakdown

There are two fundamental reasons for so many logic bugs: lack of a golden reference model and lack of a comprehensive functional coverage metric. First, there are multiple specification models above the RTL level (functional model, timing model, verification model, and so on). The consistency of these models is a major concern due to lack of a golden reference model. Second, the design verification problem is further aggravated due to lack of a functional coverage metric that can be used to determine the coverage of the microarchitectural features, as well as the quality of functional validation. Several coverage measures are commonly used during design validation, such as code coverage, FSM coverage, and so on. Unfortunately, these measures do not have any direct relationship to the functionality of the design. For example, in the case of a pipelined processor, none of these measures determine if all possible interactions of hazards, stalls and exceptions are tested.

This paper presents a validation methodology that addresses the two fundamental challenges mentioned above. Section 2 outlines traditional techniques used for validation of programmable architectures. A brief overview of our validation methodology is presented in Section 3 followed by a case study in Section 4. Finally, Section 5 concludes the paper.

2 Traditional Validation Flow

Figure 5 shows a traditional architecture validation flow. In the current validation methodology, the architect prepares an informal specification of the programmable architectures in the form of an English document. The logic designer implements the modules at the register-transfer level (RTL). The validation effort tries to uncover two types of faults: architectural flaws and implementation bugs. Validation is performed at different levels of abstraction to capture these faults. For example, architecture-level modeling (*HLM* in Figure 5) and instruction-set simulation is used to estimate performance as well as verify the functional behavior of the architecture. A combination of simulation techniques and formal methods are used to uncover implementation bugs in the *RTL design*.

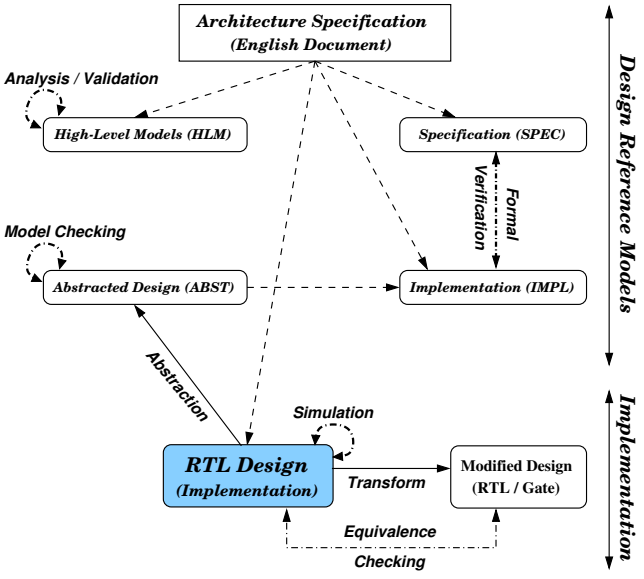

Figure 5. Traditional validation flow

Simulation using random (or directed-random) testcases [1, 4, 13, 29, 33] is the most widely used form of microprocessor validation. It is not possible to apply formal techniques directly on million-gate designs. For example, model checking is typically applied on the high-level description of the design (*ABST* in Figure 5) abstracted from the RTL implementation [9, 14]. Traditional formal verification is performed by describing the system using a formal language [3, 5, 10, 28, 30, 32, 34]. The specification (*SPEC* in Figure 5) for the formal verification is derived from the architecture description. The implementation (*IMPL* in Figure 5) for the formal verification can be derived either from the architecture specification or from the abstracted design. In current practice, the validated *RTL design* is used as a golden reference model for future design modifications. For example, when design transformations (including synthesis) are applied on the *RTL design*, the modified design (RTL/Gate) is validated against the golden *RTL design* using equivalence checking.

A significant bottleneck in these validation techniques is the lack of a golden reference model above RTL level. A typical design validation methodology contains multiple reference models depending on the abstraction level and verification activity such as functional models, timing models, formal models, models abstracted from RTL implementation, and so on. The presence of multiple reference models raises an important question - how do we maintain consistency between so many reference models?

3 ADL-driven Validation Methodology

We propose the use of a single specification to automatically generate necessary reference models. Currently the design methodology for programmable architectures typically starts with an English specification. However, it is not possible to perform any automated analysis or model synthesis on a design specified using a natural language. We propose the use of an Architecture Description Language (ADL) to capture the design specification. Figure 6 shows our ADL-driven validation methodology. The methodology has four important steps: architecture specification, validation of specification, executable (reference) model generation, and implementation (RTL design) validation.

3.1 Architecture Specification

The first step 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.

Many formal and semi-formal specification languages for describing software and hardware designs have been proposed over the years. The languages range in expressiveness, and their different levels of granularity determine their appropriateness for different applications.

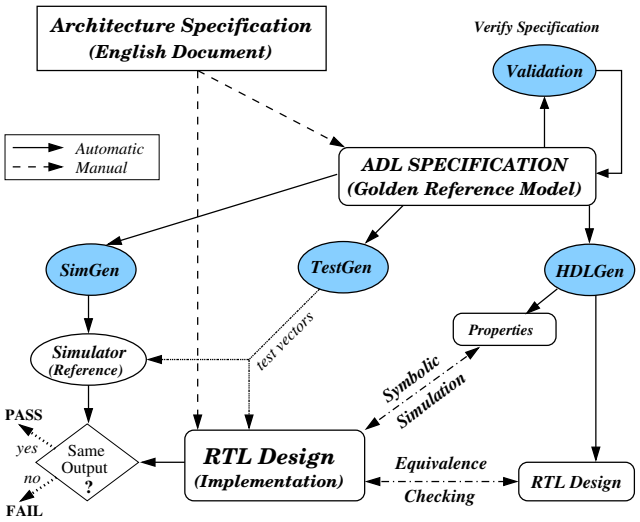

Figure 6. Proposed validation methodology

We use the EXPRESSION ADL [7] to specify programmable architectures. It is important to note that the validation techniques presented in this paper can use any existing ADL that captures both structure (components and connectivity) and behavior (instruction-set description) of the programmable architectures. The language independence allows our validation methodology to be applicable using a wide variety of specification languages.

3.2 Validation of Specification

The next step is to verify the specification to ensure the correctness of the architecture specified. One of the major challenges in validating the ADL specification of programmable architectures 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.

We have developed validation techniques to ensure that the architectural specification is well formed by analyzing both static and dynamic behaviors of the specified architecture. Figure 7 shows the flow for verifying the ADL specification. The graph model as well as the FSM model of the architecture are generated from the specification. We have developed algorithms to verify several architectural properties, such as connectedness, false pipeline and data-transfer paths, completeness, and finiteness [17, 21, 27]. The dynamic behavior is

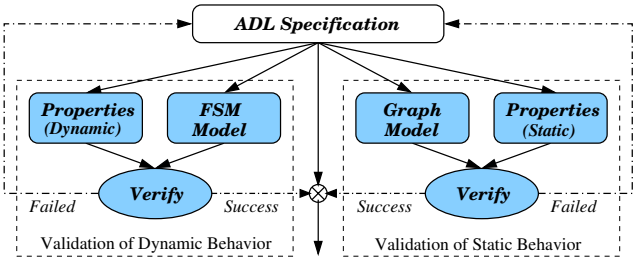

Figure 7. Validation of Specification

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 [16, 23, 26]. The 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 specification of the architecture.

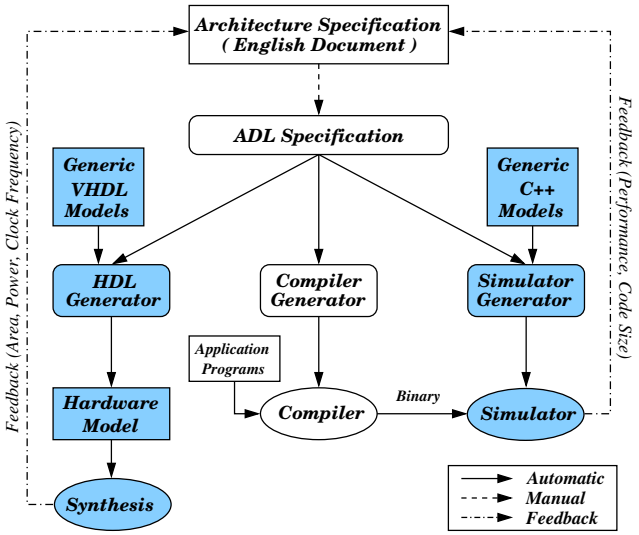

Figure 8. Simulator and Hardware Generation

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 validated ADL specification is used as a golden reference model for generating various executable models such as simulators and hardware implementations.

3.3 Executable Model Generation

A major challenge in a top-down validation methodology is the ability to generate executable models from a single 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 have 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 the necessary generic functions, sub-functions, and computational environment needed to capture a wide variety of programmable architectures [22].

The functional abstraction technique enables generation of models for simulation [22] and hardware generation [24] as shown in Figure 8. The generated models can be used for both functional validation and design space exploration. The goal of the exploration is to find the best possible architecture for a given set of application programs under various design constraints such as area, power, and performance [24, 25].

3.4 RTL Design Validation

We have explored two validation scenarios using the generated models: design validation using equivalence checking and test generation for functional validation. The generated hardware is used as a reference model for verifying the hand-written implementation (RTL Design) using a combination of symbolic simulation and equivalence checking [20]. Figure 9 shows the validation flow. To verify that the implementation satisfies certain properties, our framework generates the intended properties and uses a symbolic simulator to perform property checking. Our framework generates synthesizable RTL description of the architecture to enable equivalence checking with the hand-written implementation.

The specification is also used to generate functional test programs based on the functional coverage of pipelined architectures [15, 19]. Figure 10 shows our graph based functional test program generation methodology. The properties are generated based on the functional coverage metric. The properties are applied at the module level using the SMV model checker [11]. The counter examples are analyzed to generate test programs. 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 [19].

The generated test programs are used during simulation of the implementation and complement the tests generated by the existing techniques such as a random test pattern generator. The generated simulator is used to compute the expected outputs for the test programs.

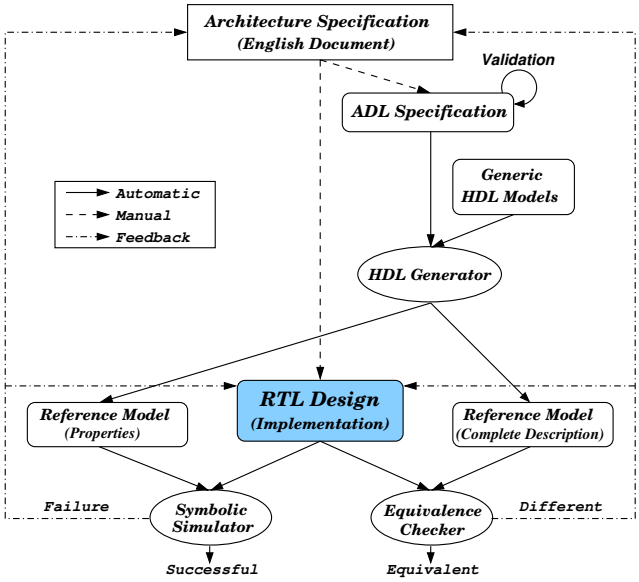

Figure 9. Implementation validation

Our experimental results demonstrate that the number of test programs generated by our approach to obtain a functional coverage is an order of magnitude less than those generated by random or constrained-random test generation techniques [18].

4 A Case Study

In a case study we successfully applied the proposed methodology to the DLX [8] processor. We have chosen the DLX processor for two reasons. First, the DLX processor has been well studied in academia and there are RTL implementations available that can be used for validation. Second, the DLX processor contains many interesting features such as fragmented pipelines and multicycle units that are representative of many commercial pipelined processor architectures (e.g., TI C6x, PowerPC and MIPS R10K).

4.1 The Architecture

Figure 11 shows the simplified version of the VLIW DLX architecture. It has five pipeline stages: fetch, decode, execute, memory (MEM), and writeback. The *execute* stage has four parallel execution paths: integer ALU, 7 stage multiplier (MUL1 - MUL7), four stage floating-point adder (FADD1 - FADD4), and multi-cycle divider (DIV). The oval boxes represent units and rectangular boxes represent storages. The solid lines represent instruction-transfer paths and dotted lines represent data-transfer paths.

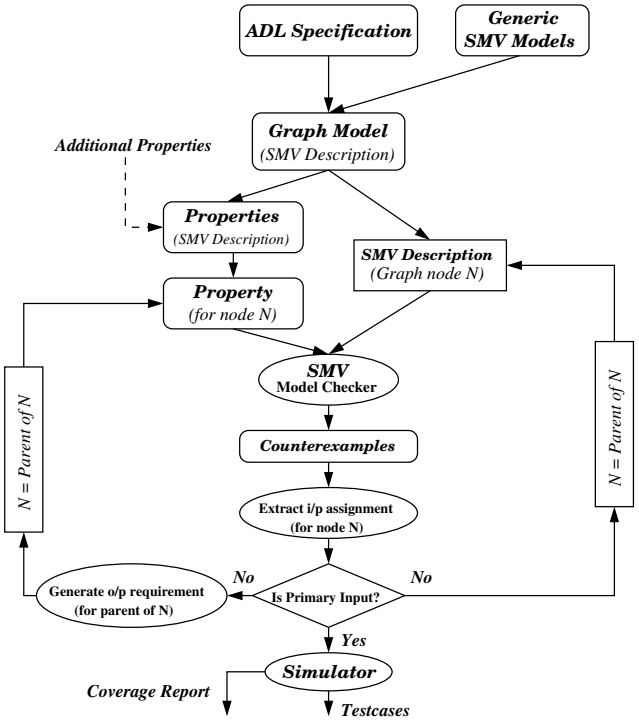

Figure 10. Functional test generation

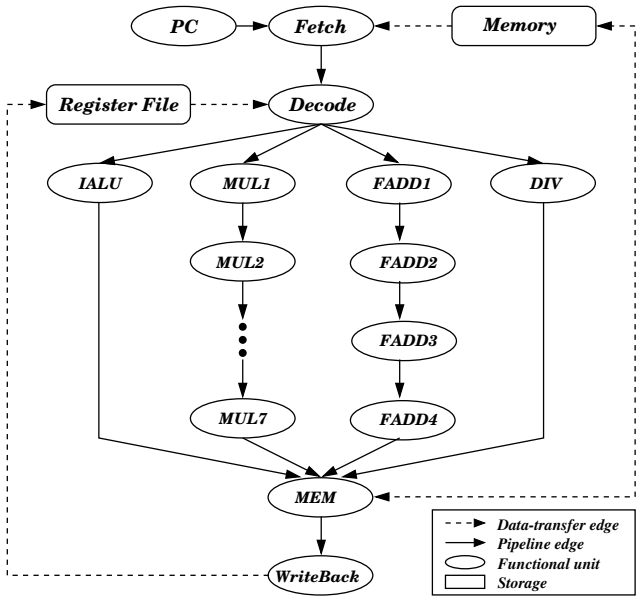

Figure 11. VLIW DLX architecture

4.2 Specification using EXPRESSION ADL

We used the EXPRESSION ADL [7] to capture the structure (components and their connectivity) and behavior (instruction-set) of the DLX processor shown in Figure 11. Figure 12(a) shows a fragment of the EXPRESSION ADL description for the five-stage pipeline {*fetch, decode, execute, memory, writeback*}. The execute stage is further described as four parallel execution paths and each execution path is described using pipeline stages. The specification also includes the description of each component and data-transfer paths.

The 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 having some common characteristics. For example, the *aluOps* in Figure 12(b) includes all the operations supported by the *IALU* 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 12(b) shows the description of the *add* and *store* operations.

4.3 Validation of Specification

The EXPRESSION ADL specification is validated for correct pipeline behavior as described in Section 3.2. The specification validation time for verifying static properties (such as connectedness, completeness, and false pipeline and data-transfer paths) is less than a second 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 for verifying the dynamic properties (such as determinism and in-order execution) is in the order of seconds.

4.4 Implementation Validation

We validated the DLX processor using equivalence checking (Section 3.4). We obtained a VHDL description of the synthesizable 32-bit RISC DLX from *eda.org* [12] and used it as the *implementation*. Our framework generated the synthesizable RTL description from the ADL specification of the DLX architecture. The generated VHDL is used as the *reference model* (specification) for the validation. To avoid memory explosion, we guided the RTL generation process such that the generated model has a structure similar to the implementation [12]. The equivalence checking process took 397 seconds on a 300 MHz Sun Ultra-250 with 1024M RAM. We also identified a bug in the computation of overflow in the implementation [12].

```
# Components specification
( FetchUnit Fetch
  (capacity 2) (timing (all 1))
  (opcodes all) (latches ...) ...
)
( ExecUnit IALU
  (capacity 1) (timing (add 1) (sub 1) ...)
  (opcodes (add sub ...)) (latches ...) ...
)
......
# Pipeline and data-transfer paths
(pipeline Fetch Decode Execute MEM WriteBack)
(Execute (parallel IALU MUL FADD DIV))
(MUL (pipeline MUL1 MUL2 MUL3 ... MUL7))
(FADD (pipeline FADD1 FADD2 FADD3 FADD4))
......
(dtpaths (WB Registers) (Registers Decode) ...)
```

(a) Structure

```
# Behavior: description of instruction set
( opgroup aluOps (add, sub, ...) )
( opgroup memOps (load, store, ...) )
......
( 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 store
  (operands (s1 reg) (s2 imm16) (s3 reg))
  (behavior M[s1 + s2] = s3)
  (format 001101 s3(25-21) s1(21-16) s2(15-0))
)
```

(b) Behavior

Figure 12. EXPRESSION Specification of VLIW DLX

4.5 Functional Test Generation

We applied our model checking based test generation technique on the DLX processor. Our framework required less than a second to generate a test program using a 333 MHz Sun UltraSPARC-II with 128M RAM and uses few thousand BDD nodes. However, if conventional techniques are used (negation of the property applied on the processor model using model checker such as SMV [11]), the test generation requires an order of magnitude increase in both time and BDD nodes as compared to our approach.

5 Conclusions

Functional validation is one of the most important problems in today's SOC design methodology. A significant bottleneck in the validation of programmable architectures is the lack of a golden reference model. Existing validation techniques employ multiple reference models and consistency of such models remains an open problem.

This paper presented an ADL-driven validation methodology for programmable architectures that uses an ADL to specify architectures. The ADL specification is used as a golden reference model and necessary executable models are generated from the specification. We have addressed three major challenges in a specification-driven validation approach: validation of specification, executable model generation, and implementation validation. This paper explored two implementation validation scenarios using the generated simulation and hardware models. First, 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. Second, we developed specification-driven test generation techniques based on the functional coverage of the pipelined architectures.

There are many challenges remaining to make this approach viable in practice. We developed a set of properties for verifying the specification. This set is by no means complete. It is important to develop a completeness criteria (to establish both necessary and sufficient conditions) for specification validation. Moreover, we considered only uni-processor based architectures. Future research needs to extend current methodology for validation of multi-processor based systems. Finally, the hardware generation and equivalence checking flow assumes that the reference and implementation models have similar structure due to the limitation of the existing equivalence checkers. 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.

References

[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] B. Bentley. High level validation of next-generation microprocessors. In *Proceedings of High Level Design Validation and Test (HLDVT)*, 2002.

[3] 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.

[4] 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.

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

[6] Gordon Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[7] 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.

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

[9] P. Ho et al. 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.

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

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

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

[13] H. Iwashita et al. Automatic test pattern generation for pipelined processors. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 580–583, 1994.

[14] 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, 2001.

[15] 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.

[16] 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.

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

[18] 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.

[19] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 182–187, 2004.

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

[21] 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.

[22] 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.

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

[24] P. Mishra et al. Synthesis-driven exploration of pipelined embedded processors. In *Proceedings of International Conference on VLSI Design*, 2004.

[25] 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.

[26] 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.

[27] P. Mishra et al. 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.

[28] 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.

[29] J. Shen et al. Functional verification of the equator MAP1000 microprocessor. In *Proceedings of Design Automation Conference (DAC)*, pages 169–174, 1999.

[30] 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.

[31] G. S. Spirakis. *Designing for 65nm and Beyond*. Keynote Address at Design Automation and Test in Europe (DATE), 2004.

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

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

[34] 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.