A Methodology for Validation of Microprocessors using Equivalence Checking

Prabhat Mishra
pmishra@cecs.uci.edu

Nikil Dutt
dutt@cecs.uci.edu

Architectures and Compilers for Embedded Systems (ACES) Laboratory
Center for Embedded Computer Systems, University of California, Irvine, CA 92697, USA
http://www.cecs.uci.edu/~aces

Abstract

As embedded systems continue to face increasingly higher performance requirements, deeply pipelined processor architectures are being employed to meet desired system performance. Validation of such processor architectures is one of the most complex and expensive tasks in the current Systems-on-Chip design process. A significant bottleneck in the validation of such systems is the lack of a golden reference model. This paper presents an Architecture Description Language (ADL) driven methodology for generating golden reference model. We use EXPRESSION ADL to capture the structure and behavior of the processor. The synthesizable Register Transfer Language (RTL) description of the architecture is generated from the ADL specification. The generated RTL description is used as a golden reference model for verifying the correctness of the implementation using equivalence checking. We applied our methodology on a RISC DLX architecture to demonstrate the usefulness of our approach.

1 Introduction

Validation of programmable embedded systems is one of the most complex and expensive tasks in the current Systems-on-Chip design process. Traditionally, architects prepare an informal specification of the microprocessor in the form of an English document. The logic designers implement the processor using Hardware Description Language (HDL). The validation engineers verifies the implementation using combination of simulation techniques and formal methods. A significant bottleneck in the validation of such systems is the lack of a golden reference model. Thus, many existing techniques ([14], [24]) employ a bottom-up approach to processor validation, where the functionality of an existing pipelined architecture is, in essence, reverse-engineered from its implementation. Our validation technique is complementary to these 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 powerful top-down approach to architecture validation.

Figure 1 shows a traditional language-driven design space exploration flow. Given a set of application programs, the goal is to find out the best possible architecture in a reasonable amount of time. The programmable embedded system (processor, coprocessor, and memory subsystem) is captured using an ADL. The ADL specification of the architecture is used to generate a software toolkit (including compiler, simulator, and assembler), and provide feedback to the designer on the quality of the architecture.

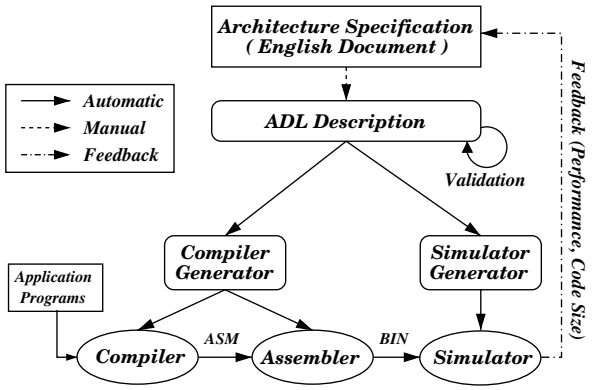

Figure 1. Language driven Design Space Exploration

An extensive body of recent work addresses ADL driven software toolkit generation and design space exploration for processor-based embedded systems, in both academia: ISDL [11], Valen-C [1], MIMOLA [19], LISA [39], EXPRESSION [12], nML [10], and industry: ARC [4], Axys [5], RADL [31], Target [36], Tensilica [37], MDES [38].

This paper presents an ADL driven methodology for verifying the correctness of the implementation using equiva-

Proceedings of the Fourth International Workshop on Microprocessor Test and Verification Common Challenges and Solutions (MTV'03)
0-7695-2045-6/03 $ 17.00 © 2003 IEEE

lence checking. The architecture is captured in EXPRESSION ADL [12]. The synthesizable HDL is generated from the ADL specification. The generated HDL description is used as a golden reference model during equivalence checking. We applied our methodology on a RISC DLX architecture to demonstrate the usefulness of our approach.

The rest of the paper is organized as follows. Section 2 briefly describes the equivalence checking technique. Section 3 presents related work addressing validation of pipelined processors. Section 4 presents our ADL-driven equivalence checking framework followed by a case study in Section 5. Finally, Section 6 concludes the paper.

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. This technique is primarily used to verify that a hardware implementation modified due to design transformations is functionally equivalent to the original implementation. The original design is assumed to be correct and known as the *reference* (golden) design. The modified design that needs to be verified against the reference design, is known as the *implementation*. The equivalence checking flow consists of four stages: *read*, *match*, *verification* and *debug*. The match and verification stages are those most impacted by design transformations [40].

During the *read* 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 2(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 2(a).

In *match* phase, the tool attempts to match, or map, compare points from the reference (golden) design to their corresponding compare point within the implementation design [3]. Two types of matching techniques are used: nonfunction (name-based) and function-based (signature analysis). Figure 2(b) shows compare point matching for a typical reference design and implementation. For better performance, the 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 will 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.

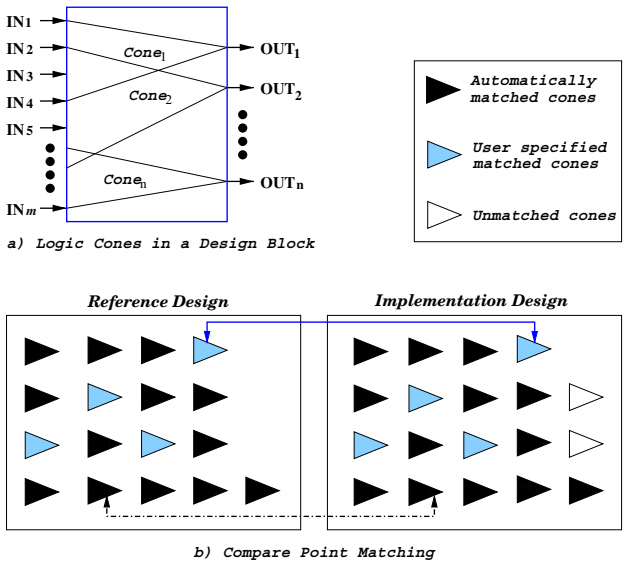

Figure 2. Matching of Compare Points between Designs

During the *verification* stage, each matched compare point is proven either functionally equivalent or non-equivalent ([9], [21]). Design transformations can impact the structure of a logic cone within 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 *debug* 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.

3 Related Work

Several approaches for formal or semi-formal verification of processors have been developed in the past. Theorem proving techniques, for example, have been successfully adapted to verify pipelined processors ([8], [29], [30], [33]). However, these approaches require a great deal of user intervention, especially for verifying control intensive designs. Hosabettu [15] proposed an approach to decompose and incrementally build the proof of correctness of pipelined microprocessors by constructing the abstraction function using completion functions.

Burch and Dill presented a technique for formally verifying pipelined processor control circuitry [6]. Their technique verifies the correctness of the implementation model

of a pipelined processor against its Instruction-Set Architecture (ISA) model based on quantifier-free logic of equality with uninterpreted functions. The technique has been extended to handle more complex pipelined architectures by several researchers [32, 41, 42]. The approach of Velev and Bryant [41] focuses on efficiently checking the commutative condition for complex microarchitectures by reducing the problem to checking equivalence of two terms in a logic with equality, and uninterpreted function symbols.

Huggins and Campenhout verified the ARM2 pipelined processor using Abstract State Machine [16]. In [20], Levitt and Olukotun presented a verification technique, called unpipelining, which repeatedly merges last two pipeline stages into one single stage, resulting in a sequential version of the processor. A framework for microprocessor correctness statements about safety that is independent of implementation representation and verification approach is presented in [2].

Ho et al. [24] extract controlled token nets from a logic design to perform efficient model checking. Jacobi [17] used a methodology to verify out-of-order pipelines by combining model checking for the verification of the pipeline control, and theorem proving for the verification of the pipeline functionality. Compositional model checking is used to verify a processor microarchitecture containing most of the features of a modern microprocessor [18].

The existing techniques attempt to formally verify the implementation of processors by comparing the pipelined implementation with its sequential (ISA) specification model, or by deriving the sequential model from the implementation. Our validation technique is complementary to these approaches. We generate synthesizable RTL from the ADL specification and use it as a golden reference model for verifying the correctness of the implementation using equivalence checking.

The industrial strength equivalence checkers (Formality [34], FormalPro [22], eCheck [28], Affirma [7], Conformal [43]) are traditionally used to check equivalence between RTL and gate level designs. It assumes that the original RTL design is golden and verifies the modified design (e.g., modified RTL or gate level design). Our technique is complementary to this methodology. Our technique ensures that the original RTL design is golden.

4 ADL-driven Validation Framework

Figure 3 shows our ADL driven validation framework. System architects develop the architecture specification document. Logic designers implement the modules to generate *RTL Design*. The first step is to specify the architecture in EXPRESSION ADL [12]. It is necessary to validate the ADL specification to ensure that the architecture is well-formed ([23], [26]). The synthesizable *hardware model* of the architecture is generated from the ADL specification. The hardware model is used as a golden reference model to verify the hand-written *RTL Design*. We use Formality [34] to check the equivalence between the hand-written *RTL Design* and the generated *hardware model*.

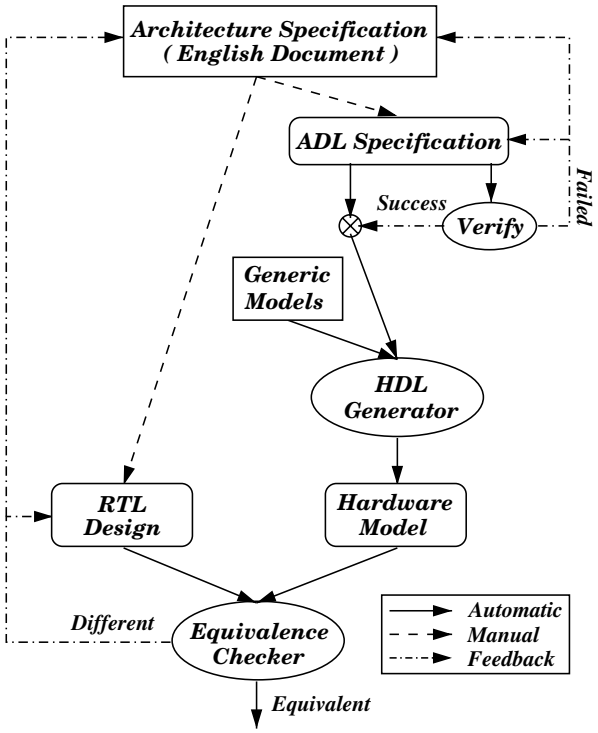

Figure 3. ADL driven Validation Framework

First, we briefly describe the EXPRESSION ADL followed by a description of the ADL-driven synthesizable HDL generation technique. Finally, we present a case study to validate DLX architecture using the generated reference model.

4.1 The EXPRESSION ADL

The EXPRESSION ADL allows automatic software toolkit generation and design space exploration of a wide range (DSP, VLIW, EPIC, Superscalar) of processors and memory subsystems. We briefly describe the key aspects of the ADL in this section. The complete reference of the language is provided in [12].

The EXPRESSION ADL captures the structure, behavior, and mapping (between structure and behavior) of the programmable architecture as shown in Figure 4.

The structure of a processor can be viewed as a graph with the components as nodes and the connectivity as the edges of the graph. It considers four types of components: *units* (e.g., ALUs), *storages* (e.g., register files), *ports*, and *connections* (e.g., buses). There are two types of edges: *pipeline edges* and *data transfer edges*. The pipeline

edges specify instruction transfer between units via pipeline latches, whereas the data transfer edges specify data transfer between components, typically between units and storages or between two storages. Each component has a list of attributes. For example, a functional unit has information regarding latches, ports, connections, opcodes, timing and capacity.

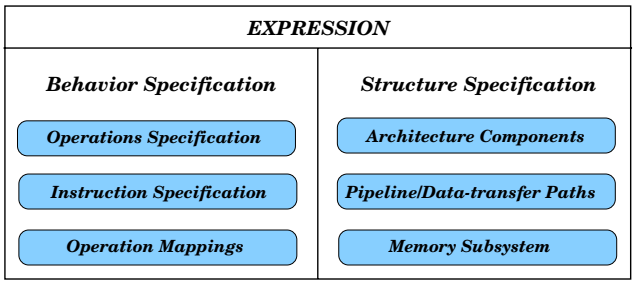

Figure 4. The EXPRESSION ADL

The behavior is organized into operation groups, with each group containing a set of operations having some common characteristics. Each operation is then described in terms of it's opcode, operands, behavior, and instruction format.

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

4.2 Synthesizable HDL Generation

The functional abstraction technique was first introduced by Mishra et al. [25] for generating simulation models for a wide variety of architectures. In this paper we have used the functional abstraction technique to automatically generate synthesizable VHDL models from the ADL specification. In fact, there is a direct relationship between generating a simulator and a hardware model: the synthesizable VHDL model is itself a simulator.

The generated HDL description consists of three major parts: instruction decoder, data-path, and control logic. We have implemented all the generic functions and subfunctions using VHDL. In this section we briefly describe how to generate three major components using the generic VHDL models. The detailed description is available in [27].

Instruction Decoder

We have implemented a generic instruction decoder that 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 operations section of the EXPRESSION ADL. The decoder extracts information regarding opcode, operands etc. from 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 (pipeline latch) to send the instruction.

Data Path

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

Control Logic

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 input instruction. For example, the local controller in an execution unit will activate the add operation if the opcode is *add*, or it will set the busy bit in case of a multi-cycle operation.

5 A Case Study

In a case study we successfully applied the proposed methodology on the DLX [13] processor. We have chosen

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 have used the synthesizable 32-bit RISC DLX implementation from University of Stuttgart [35].

The EXPRESSION ADL captures the structure and behavior of the DLX architecture. The ADL specification is validated to ensure that the architecture is well-formed ([23], [26]). Synthesizable HDL models are generated from this specification. The generated HDL description is used as a reference model. The RISC DLX from University of Stuttgart [35] is used as an implementation. We have used Synopsys Formality equivalence checker [34] to verify the implementation against the generated golden RTL.

The basic idea is simple. Irrespective of the implementation style, the equivalence checker will be able to verify the design based on the correct behavior in the reference model. For example, our HDL generation framework generates *32-bit adder* module that uses carry-look-ahead principle. The equivalence checker [34] will return *success* for the correct *32-bit adder* implementation that uses ripple-carry adder principle. The equivalence checking process took four seconds for this adder example on a 296 MHz Sun Ultra-250 with 1024M RAM.

Similarly, we generated structural model of a 32x32 register-file and used it as a reference model to verify the behavioral register-file implementation of the RISC DLX [35]. The equivalence checking process took 432 seconds for this example on a 296 MHz Sun Ultra-250 with 1024M RAM. The majority of this time (347 seconds) is spent in the elaboration (linking) phase of the behavioral implementation.

Our framework generated synthesizable RTL for 32-bit RISC DLX that supports signed operations. We guided the RTL generation process to have similar structure as in the implementation [35]. The equivalence checking process took 397 seconds on a 296 MHz Sun Ultra-250 with 1024M RAM.

6 Summary

We have presented an architecture description language driven validation framework for microprocessors. The EXPRESSION ADL captures the structure and the behavior of the architecture. The synthesizable HDL description is generated from the ADL specification using the functional abstraction technique. The generated HDL description is used as a golden reference model to verify the correctness of the implementation using equivalence checking. We applied our methodology on a RISC DLX architecture to demonstrate the usefulness of our approach.

Currently, we are able to verify designs where the reference model has similar structure as the implementation. Our future work will focus on improving this methodology for verifying designs without prior knowledge of the implementation style.

7 Acknowledgments

This work was partially supported by NSF grants CCR-0203813 and CCR-0205712. We would like to acknowledge the members of the ACES laboratory for their inputs.

References

[1] A. Inoue et al. A Programming Language for Processor Based Embedded Systems. In *Proc. of Asia Pacific Conference on Hardware Description Languages (APCHDL)*, pages 89–94, 1998.

[2] M. Aagaard, B. Cook, N. Day, and R. Jones. A Framework for Microprocessor Correctness Statements. In T. Margaria and T. Melham, editor, *Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *LNCS*, pages 433–448. Springer-Verlag, 2001.

[3] D. Anastasakis, R. Damiano, H. Ma, and T. Stanion. A Practical and Efficient Method for Compare-Point Matching. In *Proc. of Design Automation Conference (DAC)*, pages 305–310, 2002.

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

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

[6] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification (CAV)*, 1994.

[7] Cadence Affirma. *http://www.cadence.com*.

[8] D. Cyrluk. Microprocessor Verification in PVS: A Methodology and Simple Example. Technical report, SRI-CSL-93-12, 1993.

[9] C. Ejik. Sequential Equivalence Checking without State Space Traversal. In *Proc. of Design Automation and Test in Europe (DATE)*, pages 618–623, 1998.

[10] M. Freericks. The nML Machine Description Formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Dept., 1993.

[11] G. Hadjiyiannis et al. ISDL: An Instruction Set Description Language for Retargetability. In *Proc. of Design Automation Conference (DAC)*, pages 299–302, 1997.

[12] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. of Design Automation and Test in Europe (DATE)*, pages 485–490, 1999.

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

[14] R. Ho, C. Yang, M. A. Horowitz, and D. Dill. Architecture Validation for Processors. In *Proc. of International Symposium on Computer Architecture (ISCA)*, 1995.

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

[16] J. Huggins and D. Campenhout. Specification and verification of pipelining in the ARM2 RISC microprocessor. 3(4):563–580, October 1998.

[17] C. Jacobi. Formal Verification of Complex Out-of-Order Pipelines by Combining Model-Checking and Theorem-Proving. In E. Brinksma and K. Larsen, editor, *Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 309–323. Springer-Verlag, 2002.

[18] R. Jhala and K. L. McMillan. Microarchitecture Verification by Compositional Model Checking. In G. Berry et al., editor, *Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 396–410. Springer-Verlag, 2001.

[19] R. Leupers and P. Marwedel. Retargetable Code Generation based on Structural Processor Descriptions. *Design Automation for Embedded Systems*, 3(1), 1998.

[20] J. Levitt and K. Olukotun. Verifying correct pipeline implementation for microprocessors. In *Proc. of International Conference on Computer-Aided Design (ICCAD)*, pages 162–169, 1997.

[21] J. Marques-Silva and T. Glass. Combinational Equivalence Checking using Satisfiability and Recursive Learning. In *Proc. of Design Automation and Test in Europe (DATE)*, pages 145–149, 1999.

[22] Mentor FormalPro. *http://www.mentor.com*.

[23] 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 *Proc. of Design Automation and Test in Europe (DATE)*, 2002.

[24] P. Ho et al. Formal Verification of Pipeline Control Using Controlled Token Nets and Abstract Interpretation. In *Proc. of International Conference on Computer-Aided Design (ICCAD)*, 1998.

[25] P. Mishra et al. Functional Abstraction driven Design Space Exploration of Heterogeneous Programmable Architectures. In *Proc. of International Symposium on System Synthesis (ISSS)*, 2001.

[26] P. Mishra et al. Automatic Modeling and Validation of Pipeline Specifications driven by an Architecture Description Language. In *Proc. of Asia South Pacific Design Automation Conference (ASPDAC) / International Conference on VLSI Design*, 2002.

[27] P. Mishra et al. Rapid Exploration of Pipelined Processors through Automatic Generation of Synthesizable RTL Models. In *Proc. of Rapid System Prototyping (RSP)*, pages 226–232, 2003.

[28] Prover eCheck. *http://www.prover.com*.

[29] J. Sawada and W. D. Hunt. Processor Verification with Precise Exceptions and Speculative Execution. In A. Hu and M. Vardi, editor, *Computer Aided Verification (CAV)*, volume 1427 of *LNCS*, pages 135–146. Springer-Verlag, 1998.

[30] J. Sawada and J. W.A. Hunt. Trace Table based Approach for Pipelined Microprocessor Verification. In O. Grumberg, editor, *Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 364–375. Springer-Verlag, 1997.

[31] C. Siska. A Processor Description Language Supporting Retargetable Multi-pipeline DSP Program Development Tools. In *Proc. of International Symposium on System Synthesis (ISSS)*, pages 31–36, 1998.

[32] J. Skakkebaek, R. Jones, and D. Dill. Formal verification of out-of-order execution using incremental flushing. In *Computer Aided Verification (CAV)*, 1998.

[33] M. Srivas and M. Bickford. Formal Verification of a Pipelined Microprocessor. In *IEEE Software*, volume 7(5), pages 52–64, 1990.

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

[35] Synthesizable DLX: Generic 32-bit RISC Processor. *http://www.eda.org/rassp/vhdl/models/processor.html*.

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

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

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

[39] V. Zivojnovic et al. LISA - Machine Description Language and Generic Machine Model for HW/SW Co-Design. In *IEEE Workshop on VLSI Signal Processing*, 1996.

[40] R. Vallelunga and O. Eralp. Interoperable Tools Ease Equivalence Checking. *EE Times*, Feb 3, 2003.

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

[42] M. N. Velev. Formal Verification of VLIW Microprocessors with Speculative Execution. In *Computer Aided Verification (CAV)*, 2000.

[43] Verplex Conformal. *http://www.verplex.com*.