Language-driven Validation of Pipelined Processors using Satisfiability Solvers

Prabhat Mishra Heon-Mo Koo Zhuo Huang

Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611, USA.
{*prabhat, hkoo, zhuang*}*@cise.ufl.edu*

Abstract

Due to increasing demand for faster computations, deeply pipelined processor architectures are being employed to meet desired system performance. Functional validation of such pipelined processors is one of the most complex and expensive tasks in the current Systems-on-Chip design methodology. While language-based validation techniques have proposed several promising ideas, many challenges remain in applying them to realistic pipelined processors. This paper describes two practical challenges in this methodology: test generation and equivalence checking. The time and resources required for test generation using the existing approaches can be extremely large for today's pipelined processors. Similarly, traditional equivalence checkers are not useful in the context of language-driven model generation and functional validation. This paper outlines our plan to address these challenges using satisfiability checking.

1 Introduction

Functional validation is a major bottleneck is pipelined processor design. Language-based processor validation techniques have received considerable research interest in recent years. These techniques use an architecture description language (ADL) such as EXPRESSION [4], LISA [36], MIMOLA [33], or nML [21] to enable various validation efforts including test generation [15, 25, 27, 33] and equivalence checking [29].

Figure 1 shows a top-down validation methodology for pipelined processors [30]. In this methodology the processor architecture is captured using an ADL such as EXPRESSION [4]. Next, the ADL specification is validated by analyzing both static and dynamic behaviors of the specified architecture. The validated specification is then used to generate various executable models including simulator and RTL models. Finally, the implementation is validated using a combination of simulation techniques and formal methods. Necessary test programs are generated from the ADL specification using model checking. These test programs are used for simulation of the implementation. Similarly, the generated RTL model is used to verify the implementation using equivalence checking.

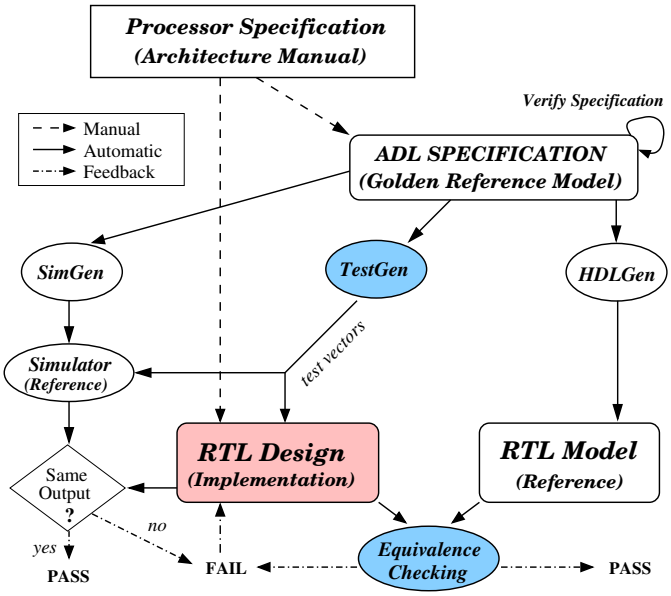

Figure 1. Language-driven Validation Methodology

While language-based validation techniques have proposed several promising ideas, many challenges remain in applying them to realistic pipelined processors. This paper describes two practical challenges in this methodology: test generation and equivalence checking. Due to lack of structural similarity between the implementation and the generate RTL model, equivalence checking is not possible for large designs. Similarly, model checking based test generation techniques are time-consuming for complex designs. This paper investigates the use of

satisfiability checking in the language-based validation methodology.

The rest of the paper is organized as follows. Section 2 presents related work addressing validation of pipelined processors. Section 3 describes our test generation methodology using SAT-based bounded model checking. Section 4 outlines our language-driven equivalence checking approach using SAT solvers followed by a case study in Section 5. Finally, Section 6 concludes the paper.

2 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 ([23], [18], [8], [34]). Burch and Dill presented a technique for formally verifying pipelined processor control circuitry [16]. This technique has been extended to handle more complex pipelined architectures by several researchers [24, 20].

Model checking based techniques have been successfully used in processor verification. Ho et al. [26] extract controlled token nets from a logic design to perform efficient model checking. Jacobi [6] 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 [32]. Industrial strength equivalence checkers have been successfully 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).

Simulation is the most widely used form of processor verification: millions of cycles are spent during simulation using a combination of random and directed test cases in traditional validation flow. Genesys [1] combines architecture knowledge, user interactive interface and testing knowledge for efficient test generation. Many techniques have been proposed for generation of directed test programs [2, 19]. Ur and Yadin [35] have presented a method for generation of assembler test programs that systematically probe the micro-architecture of a PowerPC processor. Iwashita et al. [12] use an FSM based processor modeling to automatically generate test programs. Campenhout et al. [7] have proposed a test generation algorithm that integrates high-level treatment of the datapath with low-level treatment of the controller. Mishra et al. [27, 28] have proposed functional test generation techniques using model checking.

3 Test Generation

Test generation using model checking is one of the most promising approaches due to its capability of automatic test generation. Mishra et al. [27] proposed a pipeline coverage driven test generation technique using model checking. However, the time and resources required for test generation using this approach can be extremely large for today's pipelined processors. It may not be possible to generate test programs when multiple pipeline stages are involved e.g., creating multiple exceptions scenarios. As a complementary technique, SAT-based bounded model checking (BMC) has given promising results. The basic idea is to restrict search space that is reachable from initial states within a fixed number (k) of transitions, called *bound*. After unwinding the model of design k times, the BMC problem is converted into a propositional satisfiability (SAT) problem. SAT solver is used to find a satisfiable assignment of variables that is converted into a counterexample. If the *bound* is known in advance, SAT-based BMC is typically more effective for falsification than UMC because search for counterexample is faster and SAT capacity reaches beyond BDD capacity [3]. However, finding *bound* is a challenging problem since the depth of counterexamples is unknown in general. We propose a method for determining the bound for each property instead of using a maximum bound for all properties.

Figure 2 shows our test generation methodology. Processor model is generated from the architecture specification. We use pipeline interaction fault model to define functional coverage [13]. Temporal logic properties are created from pipeline interaction faults based on the specification. We have developed a procedure for determining a bound for each property. Processor model, negated property, and the bound are applied to SAT-based BMC to generate a test program. Based on the coverage criteria, more properties can be added. We use design and property decompositions to further improve the performance of test generation. In this section we briefly outline the three important steps in SAT-based BMC: property generation, determination of bound for each property, and design decomposition. The detailed description of our test generation technique using SAT-based BMC is available in [14].

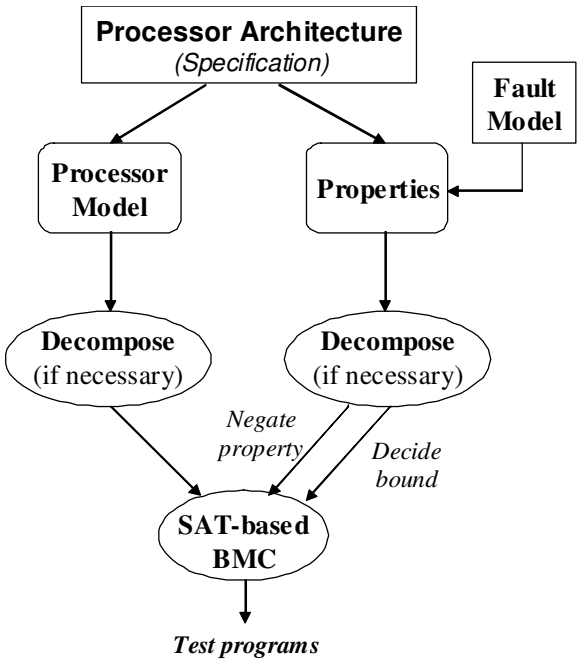

Figure 2. Test Program Generation Methodology

3.1 Property Generation

A pipeline interaction fault is converted into a property $F(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$ that combines activities over n modules using logical *AND* operator, where F is a temporal operator (*eventually*) and p_i is described as $(module_i.activity)$. The property is true if $(p_1 \wedge p_2 \wedge \ldots \wedge p_n)$ becomes true at any time step. The negation of the property, $G(\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_n)$, becomes true if any of $p_1, p_2, \ldots,$ *or* p_n is not true over all time steps.

3.2 Determination of Bound

The longest computation path in the pipeline corresponds to the bound to generate tests for all interaction scenarios. For example, in the MIPS processor [17], the maximum bound is determined by the length of {FE→ DE→ IALU→ MEM→ Cache→ MM → Cache→ MEM→ WB} if cache miss takes more time than any other pipeline paths. However, this bound is over-conservative in most test scenarios because a lot of interactions do not include this longest path. Therefore, using bound for each interaction is more efficient for test generation. The bound for each interaction fault is determined by the longest temporal distance from the Fetch unit to the nodes under consideration.

3.3 Design Decompositions

Design decompositions can be used to further improve test generation performance. We consider only two design partitioning techniques: vertical (path-level) partitioning and horizontal (stage-level) partitioning. For example, the integer-ALU pipeline path {Fetch, Decode, IALU, Mem, WriteBack} in the MIPS processor [17] is treated as one path level partition. Horizontal partitioning cuts off the descendent of the nodes under consideration because the descendent nodes do not affect the counterexample generation of property. These methods are similar to the cone of influence adopted in model checking tools. Depending on the properties, other forms of decompositions may be useful.

4 Equivalence Checking

Language-driven equivalence checking flow [29] is a promising approach but has one limitation: the structure of the generated hardware model (reference) needs to be similar to that of the processor implementation. This requirement is primarily due to the limitation of the traditional equivalence checkers. The equivalence checkers assume structural similarity and uses the similarity to reduce the complexity. Equivalence checking is not possible if the reference and implementation designs are large and drastically different. In reality, the implementation goes through numerous optimizations to improve various design parameters such as cost, area, power and performance. As a result, the final implementation may not have the similar structure as intended in the original specification, whereas the generated hardware model has the same structure as in the specification. In other words, traditional equivalence checkers are not useful in the context of language-driven model generation and functional validation.

We plan to perform equivalence checking using SAT solvers. SAT solvers along with BDD and ATPG based techniques have been successfully used in the context of hardware verification [9, 11]. Initial studies with our SAT-based combinational equivalence checker show some promising results by exploiting the structural similarities of the designs [38]. In the remainder of this section we describe our efforts in using SAT solvers for combinational as well as sequential equivalence checking.

4.1 Combinational Equivalence Checking

SAT solvers are widely used for the combinational equivalence checking (CEC) of circuits. The existing researches [9, 31, 38] use structural information to reduce the search space and improve the performance.

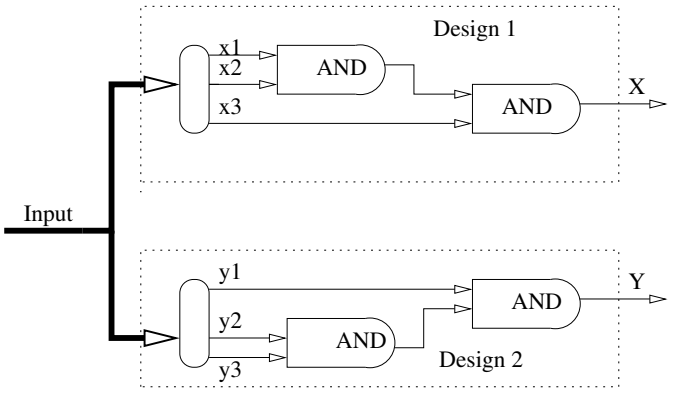

Figure 3. An example of structural similarity

Figure 3 shows a simple example of structural similarity between two designs. In Figure 3, if we can prove that x_i and y_i always have the same value, we can easily show that X and Y will always have the same value. The equivalence between $< x_i, y_i >$ can be observed from the design manually or by the CEC tool. We designed a combinational equivalence checker based on SAT solver [38] that can exploit the structural similarities between the designs automatically. The procedure with simulation, and then forms clusters based on similar signals, and finally checks whether two signals in a cluster are equivalent or not. The experimental results are shown in Table 1.

4.2 Sequential Equivalence Checking

The problem of sequential equivalence checking (SEC) is more difficult than CEC. Pipelined processors are extremely complex sequential designs. Many researches in SEC [5] tries to exploit the structural information to improve the performance. Seq-SAT [11] is one of the successful sequential SAT tool. It performs state reduction to reduce the searching time. Seq-SAT can perform SEC for two pipelined designs when the designs are small.

There are various difficulties in SEC. First, it is difficult to model the equivalence between two pipelined designs with different number of stages. For example, if one design takes 4 cycles to finish one instruction and the other design needs 5 cycles, it is very difficult to check the signals and conclude whether the two designs are equivalent or not. The problem will be more difficult for a real-life pipelined design with properties such as multi-issue, out-of-order execution and so on. Second, it is more difficult to find structural similarity in SEC than CEC, especially when this kind of similarity may happen between signals from different designs in different cycles. For example, the value of a signal X in design A at cycle k may always be equal to the value of another signal Y in design B at cycle *(k+1)*. This is a common scenario of two designs with different number of pipeline stages. It is necessary to handle such scenarios to be able to perform sequential equivalence checking on pipelined processors.

5 A Case Study

This section presents our experimental results for test generation as well as equivalence checking using SAT solvers.

5.1 Test Generation Results

We applied our methodology on a simplified MIPS architecture [17]. For our experiments, we used Cadence SMV [37] as a model checker and zChaff [22] as a SAT solver. We used 16 16-bit registers in the implementation of the register file. All the experiments were run on a 1 GHz Sun UltraSparc with 8G RAM.

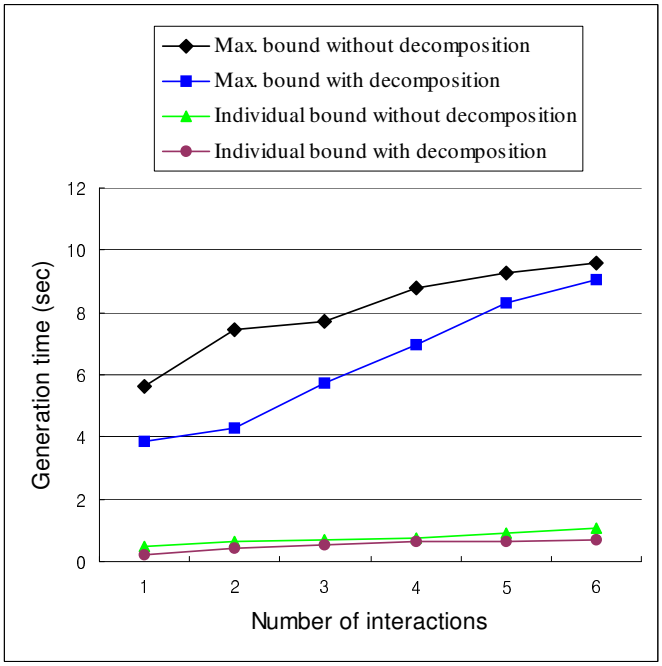

Figure 4. Comparison of Test Generation Methods

Figure 4 shows average test generation time for different module interactions by using a maximum bound for all counterexamples and each bound for each counterexample. It is important to note that unbounded model checking could not generate any test program (counterexample) in this case due to state space explosion (out of memory) problem. The X-axis in Figure 4 represents the number of interactions considered for that particular test generation

instance, and Y-axis represents the required test generation time in seconds. We used the maximum bound of 45 assuming that the longest path is taken by memory operations i.e., the sum of the IALU pipeline path length (5) and data-transfer path length (40). The figure demonstrates two important aspects. First, use of specific bound for each property improves test generation time by 90% compared to using a maximum bound in bounded model checking. Second, it is possible to achieve further improvement in test generation time by applying design and property decompositions.

5.2 Equivalence Checking Results

Table 1 shows the effect of exploiting structural similarity in CEC. The first column shows the designs from ISCAS85 benchmark suite [10]. The second column indicates the equivalence checking time without using structural similarity (SS). The last column presents the total time including determination of structural similarity and equivalence checking. All the reported time are in seconds. As expected, the equivalence checking time is drastically reduced when the structural similarity is used.

Table 1. Equivalence Checking(EC) with/without Structural Similarity(SS)

Design	EC w/o SS (seconds)	EC with SS (seconds)		
		SS time	EC with SS	Total time
C432	0.2	0.27	0.01	0.28
C499	8.05	0.72	0.3	1.02
C1355	26.13	4.54	0.3	4.84
C1908	31.15	7.74	0.2	7.94
C3540	2761.83	32.25	0.44	32.69

We used Seq-SAT [11] to test whether two simple pipelined designs are equivalent. The instruction set of each design includes NOP, AND, OR, NOT, and LOAD. Both designs have five pipeline stages: read instruction, decode, read operands, execute, and write back. However, one design has a separate path for LOAD instruction.

Table 2 shows our experimental results using Seq-SAT. The first column shows the number of registers used in each design. Similarly, the second column shows the length of each register. The third column indicates the total number of SAT variables. The last column presents the equivalence checking time. As expected the equivalence checking time is increasing drastically with the increase in design complexity. Therefore, it is necessary to exploit the structural similarity to improve the performance of sequential equivalence checking.

Table 2. Sequential Equivalence Checking for Simple Pipeline Designs

# of Registers	Reg. Length	SAT variables	Time (sec)
4	8	1469	4.78
4	16	2733	36.82
4	32	5261	19725.1
8	8	2709	26.89
8	16	5125	$>$ 20 hrs
16	4	2993	17.42
16	8	5433	$>$ 20 hrs
32	4	6173	64487.7
32	8	11429	$>$ 20 hrs

6 Conclusions

Functional verification is widely acknowledged as a major bottleneck in pipelined processor design. Language-based processor validation techniques have received considerable research interest in recent years. This paper discussed various challenges in language-based validation including test generation and equivalence checking, and studied the use of SAT solvers to address these challenges. Our experimental results had shown that SAT-based bounded model checking performs well for finding shallow counterexamples for test generation. Similarly, our initial studies had shown encouraging results for equivalence checking. Our future research includes development of efficient sequential equivalence checkers that can handle large designs without significant structural similarity.

7 Acknowledgments

We would like to thank Prof. Li-C Wang and Feng Lu of University of California, Santa Barbara for giving us the opportunity to use the Seq-SAT tool.

References

[1] A. Adir et al. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Design & Test of Computers*, 21(2):84–93, 2004.

[2] A. Aharon et al. Test program generation for functional verification of PowerPC processors in IBM. *DAC*, 1995.

[3] A. Biere, A. Cimatti, and E. M. Clarke. Bounded model checking. *Advances in Computers*, 58, 2003.

[4] A. Halambi et al. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. *DATE*, 1999.

[5] C. Eijk. Sequential equivalence checking based on structural similarities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(7):814–819, 2000.

[6] C. Jacobi. Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving. *CAV*, 2002.

[7] D. Campenhout and T. Mudge and J. Hayes. High-level test generation for design verification of pipelined microprocessors. *DAC*, 1999.

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

[9] E. Goldberg and M. Prasad and R. Brayton. Using SAT for combinational equivalence checking. *DATE*, 2001.

[10] F. Brglez, and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits. *ISCAS*, 1985.

[11] F. Lu et al. An efficient sequential sat solver with improved search strategies. *DATE*, 2005.

[12] H. Iwashita and S. Kowatari and T. Nakata and F. Hirose. Automatic test program generation for pipelined processors. *(ICCAD)*, 1994.

[13] H. Koo and P. Mishra. Functional test generation using property decompositions for validation of pipelined processors. *DATE*, 2006.

[14] H. Koo and P. Mishra. Test generation using SAT-based bounded model checking for validation of pipelined processors. *GLSVLSI*, 2006.

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

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

[17] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. 2003.

[18] J. Sawada and W. D. Hunt. Processor verification with precise exceptions and speculative execution. *CAV*, 1998.

[19] J. Shen et al. Functional verification of the equator MAP1000 microprocessor. *DAC*, 1999.

[20] J. Skakkebaek and R. Jones and D. Dill. Formal verification of out-of-order execution using incremental flushing. *CAV*, 1998.

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

[22] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik. Chaff: Engineering an efficient SAT solver. *DAC*, 2001.

[23] M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, volume 7(5), 1990.

[24] M. Velev and R. Bryant. Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction. *DAC*, 2000.

[25] O. Luthje. A methodology for automated test generation for LISA processor models. *SASIMI*, 2004.

[26] P. Ho and A. Isles and T. Kam. Formal verification of pipeline control using controlled token nets and abstract interpretation. *ICCAD*, 1998.

[27] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. *DATE*, 2004.

[28] P. Mishra and N. Dutt. Functional coverage driven test generation for validation of pipelined processors. *DATE*, 2005.

[29] P. Mishra and N. Dutt and N. Krishnamurthy and M. Abadir. A top-down methodology for validation of microprocessors. *IEEE Design & Test of Computers*, 21(2):122–131, 2004.

[30] Prabhat Mishra and Nikil Dutt. *Functional Verification of Programmable Embedded Architectures: A Top-Down Approach*. Springer, 2005.

[31] R. Arora and M. Hsiao. Using global structural relationships of signals to accelerate sat-based combinational equivalence checking. *Journal of Universal Computer Science*, 10(12):1597–1628, December 2004.

[32] R. Jhala and K. L. McMillan. Microarchitecture verification by compositional model checking. *CAV*, 2001.

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

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

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

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

[37] www-cad.eecs.berkeley.edu/~kenmcmil/smv. *SMV*.

[38] Z. Huang and P. Mishra. SAT-based combinational equivalence checking. Technical Report CISE 05-007, University of Florida, 2005.