Automatic Verification of Safety and Liveness for XScale-Like Processor Models Using WEB-Refinements

Panagiotis (Pete) Manolios
College of Computing
Georgia Institute of Technology
manolios@cc.gatech.edu

Sudarshan K. Srinivasan
School of Electrical & Computer Engineering
Georgia Institute of Technology
darshan@ece.gatech.edu

Abstract

We show how to automatically verify that a complex XScale-like pipelined machine model is a WEB-refinement of an instruction set architecture model, which implies that the machines satisfy the same safety and liveness properties. Automation is achieved by reducing the WEB-refinement proof obligation to a formula in the logic of Counter arithmetic with Lambda expressions and Uninterpreted functions (CLU). We use UCLID to transform the resulting CLU formula into a CNF formula, which is then checked with a SAT solver. We define several XScale-like models with out of order completion, including models with precise exceptions, branch prediction, and interrupts. We use two types of refinement maps. In one, flushing is used to map pipelined machine states to instruction set architecture states; in the other, we use the commitment approach, which is the dual of flushing, since partially completed instructions are invalidated. We present experimental results for all the machines modeled, including verification times. For our application, we found that the SAT solver Siege provides superior performance over Chaff and that the amount of time spent proving liveness when using the commitment approach is less than 1% of the overall verification time, whereas when flushing is employed, the liveness proof accounts for about 10% of the verification time.

1. Introduction

We show how to automatically and efficiently verify safety and liveness properties of complex XScale-like pipelined machine models. Verification entails constructing a WEB-refinement proof, which implies that, relative to a refinement map, a pipelined machine has exactly the same infinite executions as the machine defined by the instruction set architecture, up to stuttering. A consequence is that the pipelined machine satisfies exactly the same CTL*\X properties satisfied by the instruction set architecture. For the types of machines we study, we can reduce the WEB-refinement proof to a statement expressible in the logic of Counter arithmetic with Lambda expressions and Uninterpreted functions (CLU), which is a decidable logic. We use the tool UCLID to transform the CLU formula into a CNF formula, which we then check with a SAT solver. We provide experimental results for eight XScale-like pipelined machine models of varying complexity and including features such as precise exceptions, branch prediction, and interrupts. Our results show that our approach is computationally efficient, as verification times for WEB-refinement proofs are only 4.3% longer than the verification times for the standard Burch and Dill type proofs.

The use of WEB-refinement for proving the correctness of pipelined machines was introduced in [12], where some simple three stage pipelined machines were verified using the ACL2 theorem proving system [9,10]. The paper also showed that the variant of the Burch and Dill notion of correctness [3] used by Sawada [19,20] can be satisfied by machines that deadlock and an argument was given that such anomalies are not possible if WEB-refinement is used. Our main contribution is to show how one can prove WEB-refinement theorems automatically and efficiently, which we accomplish by defining rank functions and refinement maps automatically. The WEB-refinement theorem contains quantifiers and involves exhibiting the existence of certain "rank" functions and we achieve automation in two steps. First, we strengthen the theorem in a way that leads to a simplified statement, expressible in the CLU logic, that holds for the examples we consider. Second, we show how to define the rank function in a general way that does not require any deep understanding of the pipelined machine and in fact is simpler to define than flushing.

The paper is organized as follows. In section 2, we provide an overview of refinement based on WEBs, the theory upon which our correctness proofs depend. In section 3, we explain how we model XScale-like processors and in section 4, we outline how we verify such models. In section 5, we report verification times and statistics for 8 processor models, some based on the flushing approach and some on the commitment approach. We compare the time taken to prove safety alone with the time taken to prove both safety and liveness and we compare the running times of the SAT solvers Siege [18] and Chaff [16] on our problems. Everything required to reproduce our results, *e.g.*, machine models, correctness statements, CNF formulas, etc., will be available on our Web pages. Related work is described in section 6, while conclusions and an outline of future work appear in section 7.

2. Refinement

In this section, we give a brief overview of refinement based on WEBs (Well-Founded Equivalence Bisimulations), the theory underlying our pipelined machine proofs. See[12,13] for a complete description.

The point of a correctness proof is to establish a meaningful relationship between ISA, a machine modeled at the instruction set architecture level and MA, a machine modeled at the microarchitecture level, a low level description which includes the pipeline. We accomplish this by first defining a *refinement map*, r, a function from MA states to ISA states; think of r as showing us how to view an MA state as an ISA state. We then prove that MA is a WEB-refinement of ISA which implies that for every pair of states w, s such that w is an MA state and $s = r(w)$, for every infinite path σ starting at s, there is a "matching" infinite path δ starting at w, and conversely. That σ and δ "match" implies that applying r to the states in δ results in a sequence that can be obtained from σ by repeating, but only finitely often, some of σ's states, as MA may require several steps before matching a single step of ISA. We note that if MA is a refinement of ISA, then the two machines satisfy the same formulas expressible in the temporal logic CTL*\ X, over the state components visible at the instruction set architecture level. CTL*\ X is a very expressive temporal logic, allowing one to express both safety and liveness properties. For example, after we prove a WEB-refinement, we can deduce that the MA machine cannot deadlock, whereas this does not necessarily follow from the usual Burch and Dill correctness proof, even when certain "liveness" theorems are proved [12].

The main component of the refinement proof for our examples consists of showing that there exists a function *rank* mapping states of MA into the natural numbers, such that for every MA state w, if we let s be $r(w)$, u be the successor of s, and v be the successor of w, then either $r(v) = u$ or we have both $r(v) = s$ and $rank(v) < rank(w)$. The second disjunct corresponds to the case where stepping from w to v does not affect the ISA visible components; that such "stuttering" cannot continue forever is assured by showing $rank(v) < rank(w)$, as the range of *rank* is well founded.

The theoretical work on WEB-refinements [13] is quite general and the main component of the theorem we prove is a stronger statement than what is required to prove WEB-refinement in a general setting; nonetheless, it is a statement that holds for the examples we consider. There is a good reason that we strengthened the WEB-refinement proof obligation: this allows us to obtain a statement expressible in CLU, after we define *rank*. The definition of *rank* depends on the definition of the refinement map r.

We use two types of refinement maps. One is based on flushing. The other, based on the commitment approach, can be thought of as the dual of flushing, since partially completed instructions are invalidated instead of completed. For the flushing approach, the *rank* of a state is essentially the number of clock cycles required to fetch a new instruction which will make it through the pipeline (to match a step of the ISA machine). For the commitment approach, the *rank* of a state is the number of clock cycles required to retire an instruction (to match a step of the ISA machine). We provide a general method for defining *rank* for both types of refinement maps. A more detailed description appears in the following sections.

3. Modeling of XScale-Like Processors

Figure 1 shows the high-level organization of the XScale-like processor model. The model is a seven stage pipeline whose stages are IF1, IF2 (2-cycle fetch), ID (instruction decode), EX (execute), MEM1, MEM2 (2-cycle memory access), and WB (write back). Five abstract instruction types are modeled including register-register, register-immediate, load, store, and branch. The branch and store instructions complete out of order with respect to the ALU instructions. This base model is extended with branch prediction, ALU exceptions and interrupts.

The branch predictor is abstracted with a state variable, *BPState* that holds the current state of the branch predictor, and 2 UFs and a UP including *NextBPState*, *PredictTarget* and *PredictDirection* that only take the *BPState* as input. *NextBPState*, *PredictDirection*, and *PredictTarget* produce as output the next state of the branch predictor, an arbitrary prediction on the direction, and an arbitrary prediction on the target of the branch, respectively. The actual direction and target of the branch are determined in EX. Mispredictions are corrected in MEM1. What is verified is the logic to correct mispredictions.

ALU exceptions are modeled with a UP that takes the same inputs as the ALU, and outputs a predicate indicating if an exception is raised. ALU exceptions are dealt with in MEM1. In case of an ALU exception, all previous instructions are squashed, the program counter is updated with the address corresponding to the ALU exception handler routine, and the PC of the excepting instruction is stored in the Exception Program Counter (EPC). A return-from-exception instruction is also implemented that restores the PC with the EPC.

Interrupts are modeled with an arbitrary interrupt state *INPState*, a UF *NextINPState* that takes INPState as input and produces the next interrupt state, and UP *IsInterrupt* that also takes *INPState* as input and produces a predicate which indicates if an interrupt is raised. Interrupts are detected in the MEM1 stage and squash all previous instructions including the instruction that caused the interrupt. We use temporal abstraction to model the behavior of interrupts. The only trace left by an interrupt is that it has modified the data memory. The PC is set to the program counter corresponding to the first instruction that was squashed by the interrupt, the data memory is modified using a UF that takes the previous data memory state as input, and the register file is not modified.

4. Verification of XScale-Like Processor Models

We prove the core theorem on the various XScale-like processor models with out of order completion, branch prediction, ALU exceptions, and interrupts. We use two different approaches, including the commitment approach and flushing, to define the refinement map. The core theorem is defined below.

$$
\begin{aligned}
& s = r(w) \quad \wedge \\
& u = \text{ISA-step}(s) \quad \wedge \\
& v = \text{MA-step}(w) \quad \wedge \\
& u \neq r(v) \quad \wedge \\
\Rightarrow & \\
& s = r(v) \quad \wedge \quad \text{rank}(v) < \text{rank}(w)
\end{aligned}
$$

In the theorem shown above *s* and *u* are ISA states, and *w* and *v* are MA states; ISA-step is a function corresponding to stepping the ISA machine once and MA-step is a function corresponding to stepping the MA machine once; *r* is the refinement map that maps MA states to ISA states; and rank is the rank function. The theorem says that if *s* is the refinement of *w*, *u* is obtained by stepping *s*, *v* is obtained by stepping *w*, and *u* is not the refinement of *v*, then *s* is the refinement of *v* and the rank of *v* is less than the rank of *w*. The proof obligation relating *s* and *v* is the safety component, and the proof obligation that rank(*v*) < rank(*w*) is the liveness component.

4.1 Commitment Approach

The commitment approach relates ISA and MA states by retaining the programmer visible components of the committed part of the MA states. A committed MA state is obtained by invalidating all the partially executed instructions in the pipeline, and rolling back the MA state to correspond with the last committed instruction. The MA states are rolled back using history variables that store the MA states corresponding to the last *n* MA steps, where *n* is the number of steps an instruction takes to be committed after it updates the

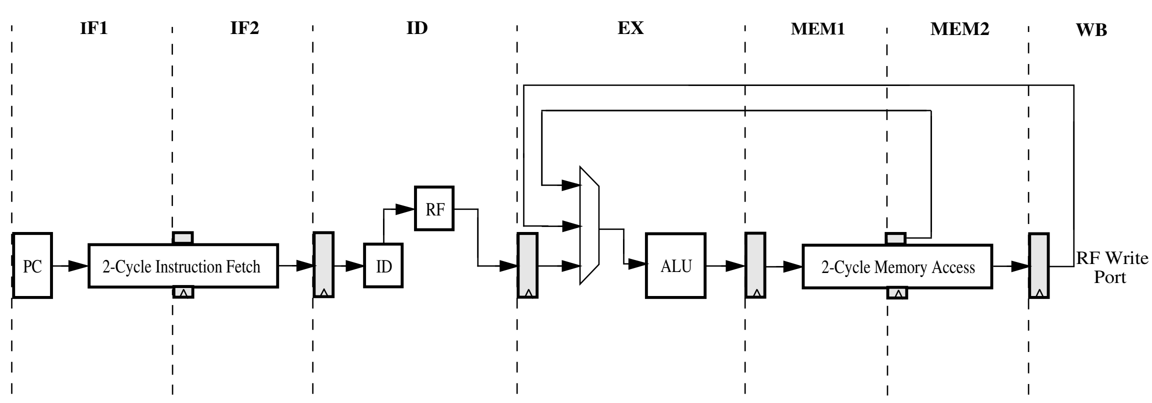

Figure 1. Pipeline organization of processor model.

MA state. The histories of the PC, data memory, and interrupt state are stored for the last 6, 2, and 2 steps, respectively. Only the current state of the register file is required as, when the instruction writes back to the register file, it is considered committed. In order to use the commitment approach, it is required that we compare ISA states only with "good" MA states, where an MA state is "good" iff it is reachable from a committed state. To check that an MA state, *w*, is "good", the committed state, *c*, corresponding to *w* is determined. State *w* is "good" if it can be reached from *c* in 0 to 6 steps. In this approach, the rank of an MA state is the number of MA-steps required to commit a new instruction. The rank is determined by checking the first valid latch that is closest to the register file. An MA state with a valid latch closer to the register file is assigned a higher rank.

4.2 Flushing Approach

The flushing approach relates MA and ISA states by flushing an MA state and comparing the resultant programmer visible components of the MA state with the ISA state, where by flushing we mean feeding the pipeline with bubbles to complete partially executed instructions without fetching any new instructions. It turns out that for a single-issue pipelined machine, the safety proof of the core WEB theorem is similar to the Burch and Dill approach [3]. In the flushing approach, the rank of an MA state, *w*, is the number of steps required to fetch a new instruction that eventually completes. The rank can be determined by stepping *w*, to obtain *v*, flushing *v*, and comparing the result with the flushed state of *w*, to check if it has made any progress. The number of steps required to make progress is the rank. The straightforward implementation of this idea requires 174 symbolic simulations, which UCLID was not able to handle. We implemented an optimized version based on the observation that stepping and flushing the MA states can be folded together so as to reduce the number of symbolic simulations. In more detail, we determine the number of steps required to flush the pipeline (by flushing it) and we set a counter to this value. The MA state is simulated for this number of steps and the rank of the MA state is the number of steps required for the latch closest to the register file to become valid.

4.3 CLU Logic

The CLU logic consists of Uninterpreted Functions (UFs) and Predicates (UPs), restricted lambda expressions, ordering, and successor and predecessor functions. Combinational logic is abstracted with UFs/UPs. The output of a UF is a term variable and a UP is a truth variable. The only property satisfied by UFs and UPs is functional consistency—when the inputs of two different instances of a UF are equal, it implies that the outputs are equal. The successor function is used to define the rank functions for the MA states. We could easily do without the successor function since the rank of a state is always less than the number of latches in the pipeline. This means that our approach is applicable even with tools that only support the logic of equality with uninterpreted functions and memories, but we find that defining rank explicitly is clearer and performance is essentially the same.

5. Results

In this section, we review our experimental results. We start with two base processor models, CXS and FXS: the prefix C indicates the use of the commitment approach for defining the refinement map and prefix F indicates the use of flushing for defining the refinement map. Both models can execute 6 basic abstract instruction types including register-register, register-immediate, branch, load, store, and return-from-exception with out of order completion. The base models are extended to implement:

1) branch prediction, designated by "-BP";
2) ALU exceptions, designated by "-EX"; and
3) interrupts, designated by "-INP".

Table 1 presents the results. We report the number of CNF variables and clauses and the verification time for both the safety proofs and the safety and liveness proofs. For the safety and liveness proofs, we also report the size of the CNF files and the verification times taken by both Siege and Chaff. The total verification time reported includes the time taken by Siege and UCLID, thus the time taken by UCLID can be obtained by subtracting the Siege column from the Total column. Siege uses a random number generator, which leads to large variations in the execution times obtained from multiple runs of the same input, thus, in order to make reasonable comparisons, every Siege entry is really the average over 10 runs and we report the standard deviations for the runs. The experiments were conducted on an Intel XEON 2.20GHz processor with an L1 cache size of 512KB.

As is clear from Table 1, Siege provides superior performance when compared to Chaff. If we divide the total running time of Chaff with Siege, we see that Siege provides a speedup of about 17 and in the case of CXS the speedup is 226. The overall cost of liveness, computed by subtracting the sum of the Safety Siege

Processor	Safety				Safety and Liveness						
	CNF Vars	CNF Clauses	Verification Time [sec]		CNF Var	CNF Clauses	CNF Size [KB]	Verification Time [sec]			
			Siege	Total				Siege	Chaff	Stdev	Total
CXS	12,930	38,215	35	38	12,495	36,925	664	29	6,552	3.4	32
CXS-BP	24,640	72,859	284	289	23,913	70,693	1,336	300	7,861	48.7	305
CXS-BP-EX	24,651	72,841	244	249	24,149	71,350	1,344	233	4,099	50.2	238
CXS-BP-EX-INP	24,669	72,880	255	261	24,478	72,322	1,368	263	3,483	34.1	269
FXS	28,505	36,925	140	154	53,441	159,010	3,096	160	796	24.4	175
FXS-BP	33,964	100,624	170	185	71,184	211,723	4,136	187	586	50.4	203
FXS-BP-EX	35,827	106,114	179	195	74,591	221,812	4,344	163	759	17.6	180
FXS-BP-EX-INP	38,711	11,4742	128	147	81,121	241,345	4,736	170	1,427	32.3	189

Table 1. Statistics for boolean correctness formula and formal verification time.

column from the sum of the Safety and Liveness Siege column and dividing by the latter is 4.6%; notice that for the commitment approach it is 0.75%, whereas it is 9.3% for the flushing approach. Finally, we note that there are cases in which the verification time for safety and liveness is less than that of liveness; in fact, the verification time for liveness alone seems to be about the same as the verification time for safety, *e.g.*, when proving liveness for CXS, Siege takes 37 seconds (this is the average of ten runs).

All machine models, correctness statements, CNF formulas, and in general everything required to reproduce our results will be available on our Web pages.

6. Related Work

We now review previous work on pipelined machine verification. A very early approach by Srivas and Bick was based on the use of skewed abstraction functions [23]. Burch and Dill showed how to automatically compute the abstraction function using flushing [3]. There are approaches based on model-checking, *e.g.*, in [14], McMillan uses compositional model-checking in conjunction with symmetry reductions. Theorem proving approaches are also popular, *e.g.*, in [19,20], Sawada uses an intermediate abstraction called MAETT to verify some safety and liveness properties of complex pipelined machines. Another approach by Hosabettu et al. uses the PVS theorem prover and the notion of completion functions [5]. Symbolic Trajectory Evaluation (STE) is used by Patankar et al. to verify a processor that is a hybrid between ARM7 and StrongARM [17]. SVC is used check the correct flow of instructions in a pipelined DLX model [15]. Abstract State Machines are used to prove the correctness of refinement steps that transform a non-pipelined ARM processor into a pipelined implementation [6]. An XScale processor model is verified using a variation of the Burch and Dill approach in [22].

This paper directly depends on previous work on decision procedures for boolean logic with equality and uninterpreted function symbols [1]. The results in [1] were further extended in [2], where a decision procedure for the logic of Counter arithmetic with Lambda expressions and Uninterpreted functions (CLU) is given. The decision procedure is implemented in UCLID, which has been used to verify out-of-order microprocessors [21].

7. Conclusions and Future Work

We show how to automatically verify safety and liveness properties of complex XScale-like pipelined machine models with a slight performance penalty over verifying safety properties alone. This is accomplished by proving a WEB-refinement theorem, which implies that the pipelined machine satisfies exactly the same CTL*\X properties satisfied by the instruction set architecture. We show how to automate the verification of the WEB-refinement theorem, which contains quantifiers and involves exhibiting the existence of certain ''rank'' functions. The automation is achieved in two steps. First, we strengthen the theorem in a way that leads to a simplified statement that holds for the examples we consider. Second, we show how to define the rank function in a general way that does not require any deep understanding of the pipelined machine; in fact, it much simpler to define the rank function than it is to define how the machine is flushed. As a result, we are left with a formula in the logic of Counter arithmetic with Lambda expressions and Uninterpreted functions and can use UCLID to obtain a CNF formula, which we then check with a SAT solver. To summarize, our main contribution is to show how WEB-refinements can be used as the basis for automatic verification of pipelined machines, resulting in both safety and liveness verification, with only a slight increase in verification times.

For future work, we are planning to explore how one can connect UCLID (any decision procedure for CLU will do) with the theorem proving system ACL2 [9,10]. This will allow us to use ACL2 for efficient simulation and advanced debugging. In addition, we plan to explore methods for verifying larger instructions sets more efficiently than is currently possible with either approach alone.

References

[1] R.E. Bryant, S. German, and M.N. Velev, "Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions," *Computer-Aided Verification (CAV '99)*, N. Halbwachs and D. Peled, eds., *LNCS* 1633, Springer-Verlag, July 1999, pp. 470-482.

[2] R.E. Bryant, S.K. Lahiri, and S.Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In E. Brinksma and K. Larsen, editors, *Computer-Aided Verification--CAV 2002}*, volume 2404 of *LNCS*, pages 78-92. Springer-Verlag, 2002.

[3] J.R. Burch, and D.L. Dill, "Automated Verification of Pipelined Microprocessor Control," *Computer-Aided Verification (CAV'94),* D.L. Dill, *ed*., LNCS 818, Springer-Verlag, June 1994, pp. 68–80.

[4] L.Clark, E.Hoffman, J.Miller, M.Biyani, Y.Liao, S.Strazdus, M.Morrow, K.Velarde, and M.Yarch, "An embedded 32-bit microprocessor core for low-power and high-performance applications, *IEEE Journal of Solid-State Circuits*, pp. 1599-1608, 2001.

[5] R.Hosabettu, M.Srivas, and G.Gopalakrishnan, "Proof of correctness of a processor with reorder buffer using the completion functions approach," In N.Halbwachs and D.Peled, editors, *Computer-Aided-Verification--CAV '99*, volume 1633 of *LNCS*. Springer-Verlag, 1999.

[6] J.K. Huggins, and D.V. Campenhout, "Specification and Verification of Pipelining in the ARM2 RISC Microprocessor," ACM Transactions on Design Automation of Electronic Systems, Vol. 3, No. 4 (October 1998), pp. 563–580.

[7] W.A. Hunt, Jr. and S.D. Johnson, editors. Formal Methods in Computer-Aided Design--FMCAD 2000, volume 1954 of *LNCS*. Springer-Verlag, 2000.

[8] M.Kaufmann, P.Manolios, and J.S. Moore, editors. "*Computer-Aided Reasoning: ACL2 Case Studies*," Kluwer Academic Publishers, June 2000.

[9] M.Kaufmann, P.Manolios, and J.S. Moore, "*Computer-Aided Reasoning: An Approach*." Kluwer Academic Publishers, July 2000.

[10] M.Kaufmann and J.S. Moore,ACL2 homepage. See URL http://www.cs.utexas.edu/users/moore/acl2.

[11] S.K. Lahiri, S.A. Seshia, and R.E. Bryant, "Modeling and Verification of Out-of-order Microprocessors using UCLID," *Formal Methods in Computer-Aided Design (FMCAD'02)*, LNCS 2517, pp. 142-159, November 2002.

[12] P.Manolios, "Correctness of pipelined machines," In Hunt and Johnson [6], pages 161-178.

[13] P.Manolios, *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, August 2001.

[14] K.L. McMillan, "Verification of an implementation of Tomasulo's algorithm by compositional model checking," In A.J. Hu and M.Y. Vardi, editors, *Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pp. 110-121. Springer-Verlag, 1998.

[15] P. Mishra, and N. Dutt, "Modeling and Verification of Pipelined Embedded Processors in the Presence of Hazards and Exceptions," IFIP WCC 2002 Stream 7 on Distributed and Parallel Embedded Systems (DIPES'02), August 2002.

[16] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *Design Automation Conference (DAC'01)*, 2001, pp. 530–535.

[17] V.A. Patankar, A. Jain, and R.E. Bryant, "Formal verification of an ARM processor," Twelfth International Conference On VLSI Design, 1999, pp. 282–287.

[18] L. Ryan, Siege homepag. See URL http://www.cs.sfu.ca/ ld loryan/personal.

[19] J.Sawada, *Formal Verification of an Advanced Pipelined Machine.* PhD thesis, University of Texas at Austin, Dec. 1999. See URL. http://www.cs.utexas.edu/users/sawada/dissertation/.

[20] J.Sawada,Verification of a simple pipelined machine model. In Kaufmann et.al. [7], pp. 137--150.

[21] S.A. Seshia, S.K. Lahiri, and R.E. Bryant, "A Hybrid SAT-Based Decision Procedure for Separation Logic with Uninterpreted Functions,". *Design Automation Conference (DAC'03)*, pp. 425-430, June 2003.

[22] S.K. Srinivasan, and M.N. Velev, "Formal Verification of an Intel XScale Processor Model with Scoreboarding, Specialized Execution Pipelines, and Imprecise Data-Memory Exceptions," *Formal Methods and Models for Codesign (MEMOCODE '03)*, June 2003, pp. 65-74.

[23] M.Srivas and M.Bick, "Formal verification of a pipelined microprocessor," *IEEE Software*, pp. 52-64, Sept. 1990.