Integrating Formal Verification into an Advanced Computer Architecture Course

Miroslav N. Velev
mvelev@ece.gatech.edu
School of Electrical and Computer Engineering
Georgia Institute of Technology, Atlanta, GA 30332, U.S.A.

Abstract. The paper presents a sequence of three projects on design and formal verification of pipelined and superscalar processors. The projects were integrated—by means of lectures and preparatory homework exercises—into an existing advanced computer architecture course taught to both undergraduate and graduate students in a way that required them to have no prior knowledge of formal methods. The first project was on design and formal verification of a 5-stage pipelined DLX processor, implementing the six basic instruction types—register-register-ALU, register-immediate-ALU, store, load, jump, and branch. The second project was on extending the processor from project one with ALU exceptions, a return-from-exception instruction, and branch prediction; each of the resulting models was formally verified. The third project was on design and formal verification of a dual-issue superscalar version of the DLX from project one. The preparatory homework problems included an exercise on design and formal verification of a staggered ALU, pipelined in the style of the integer ALUs in the Intel Pentium 4. The processors were described in the high-level hardware description language AbsHDL that allows the students to ignore the bit widths of word-level values and the internal implementations of functional units and memories, while focusing entirely on the logic that controls the pipelined or superscalar execution. The formal verification tool flow included the term-level symbolic simulator TLSim, the decision procedure EVC, and an efficient SAT-checker; this tool flow—combined with the same abstraction techniques for defining processors with exceptions and branch prediction, as used in the projects—was applied at Motorola to formally verify a model of the M•CORE processor, and detected bugs. The course went through two iterations—offered at the Georgia Institute of Technology in the summer and fall of 2002—and was taught to 67 students, 25 of whom were undergraduates.

1. Introduction

Verification is increasingly becoming the bottleneck in the design of state-of-the-art computer systems, with up to 70% of the engineering effort spent on verifying a new product[1]. The increased complexity of new microprocessors leads to more errors—Bentley[2] reported a 350% increase in the number of bugs detected in the Intel Pentium 4[3], compared to those detected in the previous architecture, the Intel Pentium Pro[4, 5]. Formal verification—the mathematical proof of correctness of hardware and software—is gaining momentum in industrial use, but has so far failed to scale for realistic microprocessors, or has required extensive manual intervention by experts—factors that have made formal verification impractical to integrate in existing computer architecture courses.

Traditionally, the principles of pipelined and superscalar execution have been taught with three approaches: 1) trace-driven simulation with existing tools[6–18]; or 2) implementation of a trace-driven simulator by using a programming language such as C, often by filling only missing sections in skeleton code provided by the instructors[19–21]; or 3) implementation of a processor using a commercial hardware description language (HDL), such as Verilog[22] or VHDL[23–24] as reported by several authors[25–28], or an academic HDL[20, 29], and then simulating it with existing tools or with an FPGA-based hardware emulation system[21, 30–34]. In the second and third approaches, the implementations are verified with test sequences provided by the instructors, and possibly with additional test sequences defined by students. However, such testing is time consuming and, most importantly, does not guarantee complete correctness—with bugs remaining undetected for years even in the skeleton code provided by the instructors[19].

In spite of the role that formal verification will play when designing computer systems in the future, only a few universities offer related courses[35]. There are many reports on the integration of formal methods into existing software engineering curricula[36–40]. However, the author knows of only one computer architecture course[19] that was previously offered at CMU in a version where the students had to model check[41] a cache coherence protocol.

This paper advocates the integration of formal verification into existing computer architecture courses, as a way to educate future microarchitects with knowledge of formal methods, and with deeper understanding of the principles of pipelined, speculative, and superscalar execution; microarchitects who are thus more productive, and capable of delivering correct new processors under aggressive time-to-market schedules. The paper presents the experience from such an integration of formal verification into an existing computer architecture course[42–43], taught to both undergraduate and graduate students with no prior knowledge of formal methods. The existing course was extended with several lectures on formal verification, with related homework problems, and with a sequence of three projects on design and formal verification of processors: 1) a single-issue pipelined DLX[44]; 2) a version with exceptions and branch prediction; and 3) a dual-issue superscalar DLX. The last project was motivated by commercial dual-issue superscalar processors, such as the Intel Pentium[45], the Alpha 21064[46], the IDT RISCore5000[47], the PowerPC 440 Core[48], the Motorola MC 68060[49], the Motorola MPC 8560[49], and the MIPS III used in the Emotion Engine chip of the Sony Playstation 2[44].

The integration of formal verification into an existing computer architecture course was made possible by a recently developed tool flow, consisting of the term-level symbolic simulator TLSim[50], the decision procedure EVC[50–53], and an efficient Boolean Satisfiability (SAT) checker, such as Chaff[54, 55] or BerkMin[56]. The pedagogical power of this tool flow is due to: 1) the immediate feedback given to students—it takes 1.5 seconds to formally verify a single-issue pipelined DLX processor[44], 7 seconds to formally verify an extension with exceptions and branch prediction, and 10 minutes to formally verify a dual-issue superscalar DLX; 2) the full detection of bugs, when a processor is formally verified; and 3) the resulting objective grading—based on complete correctness, as opposed to just passing tests provided by the instructor.

Historically, every time the design process was shifted to a higher level of abstraction, productivity increased. That was the case when microprocessor designers moved from the transistor level to the gate level, and later to the register-transfer level, relying on Electronic Design Auto-

mation (EDA) tools to automatically bridge the gap between these levels. Similarly, that was the case when programmers adopted high-level programming languages such as FORTRAN and C, relying on compilers for translation to assembly code. The high-level hardware description language AbsHDL differs from commercial HDLs such as Verilog and VHDL in that the bit widths of word-level values are not specified, and neither are the implementations of functional units and memories. That allows the microprocessor designers to focus entirely on partitioning the functionality among pipeline stages and on defining the logic to control the pipelined or superscalar execution. Most importantly, this high-level definition of processors, coupled with certain modeling restrictions (see Section 3), allows the efficient formal verification of the pipelined designs. The assumption is that the bit-level descriptions of functional units and memories are formally verified separately, and will be added by EDA tools later, when an AbsHDL processor is automatically translated to a bit-level synthesizable description, e.g., in Verilog or VHDL.

The rest of the paper is organized as follows. Section 2 defines the high-level hardware description language AbsHDL, used in the projects. Section 3 summarizes the formal verification background, and presents the tool flow. Section 4 describes the three projects, and Section 5 discusses their integration into an existing advanced computer architecture course. Section 6 analyzes the results and the design bugs made by students when implementing the projects. Section 7 reviews related work. Section 8 concludes the paper and outlines directions for future work.

2. The High-Level Hardware Description Language AbsHDL

The syntax of AbsHDL will be illustrated with the 3-stage pipelined processor, pipe3, shown in Figure 1. That processor has a combined instruction fetch and decode stage (IFD), an execute stage (EX), and a write-back stage (WB). It can execute only ALU instructions with a single data operand. Read-after-write hazards are avoided with one level of forwarding. The AbsHDL definition of pipe3 is shown in Figure 2. We will use extension `.abs` for files in AbsHDL.

An AbsHDL processor description begins with declaration of signals (see Figure 2). Bit-level signals are declared with the keyword `bit`; word-level signals with the keyword `term`. Signals that are primary inputs, e.g., phase clocks, are additionally declared with the keyword `input`. The language has constructs for basic logic gates—`and`, `or`, `not`, `mux`—such that `and` and `or` gates can have multiple inputs. Equality comparators are gates of type `=`, e.g., `RegsEqual = (= IFD_EX_SrcReg EX_WB_DestReg)` in the EX stage, where the equals sign before the left parenthesis designates an assignment. Gates that are not of the above types are called *uninterpreted functions* if the output is a word-level signal—e.g., `sequentialPC = (PCAdder PC)` and `Result = (ALU IFD_EX_Op EX_Data1)` in Figure2—but *uninterpreted predicates* if the output is a bit-level signal. Uninterpreted functions and uninterpreted predicates are used to abstract the implementations of combinational functional units. In the two examples above, `PCAdder` and `ALU` are uninterpreted functions that abstract, respectively, the adder for incrementing the PC and the ALU in pipe3. We can use an uninterpreted predicate to abstract a functional unit that decides whether to take a conditional branch, or to abstract a functional unit that indicates whether an ALU exception is raised. We can implement a finite state machine (FSM) to model the behavior of a multicycle functional unit[51].

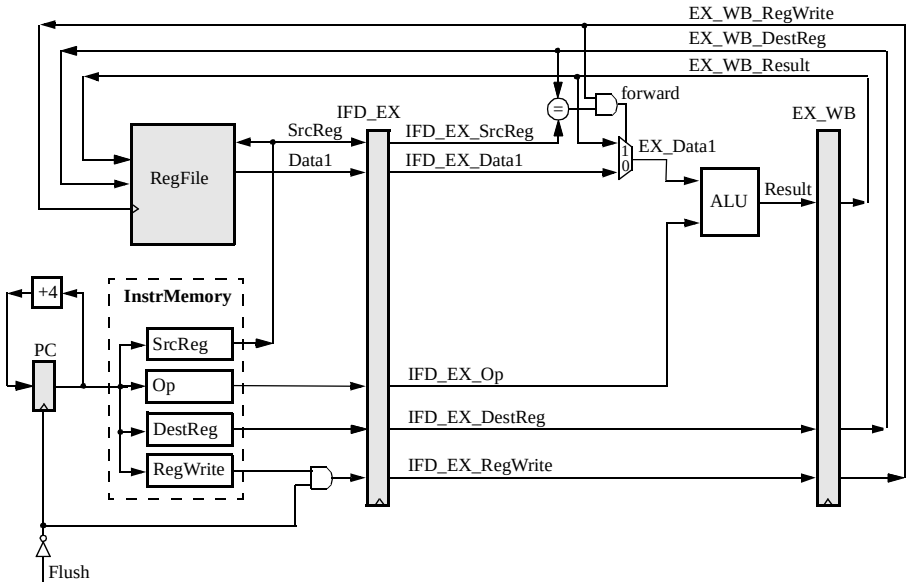


Figure 1. Block diagram of the 3-stage pipelined processor pipe3.

AbsHDL has constructs for latches and memories, defined with the keywords `latch` and `memory`, respectively. Both can have input and/or output ports, defined with the keywords `inport` and `outport`, respectively. Input ports of latches have an enable signal, which has to be high for a write operation to take place at that port, and a list (enclosed in parentheses) of input data signals that provide the values to be written to the latch. Similarly, output ports of latches have an enable signal, which has to be high for a read operation to take place at that port, and a list of output data signals that will get the values stored in the latch. An output data signal can get values only from input data signals that appear in the same position in the list of data signals for an input port in the same latch. Memories are defined similarly, except that ports additionally have an address input that is listed right after the enable input—see memory `RegFile` in Figure 2.

The correct instruction semantics are defined by the instruction set architecture (ISA) and are modeled with a non-pipelined specification processor built from the same uninterpreted functions, uninterpreted predicates, and architectural state elements (the PC and the Register File in pipe3) as the pipelined implementation. Since the specification is non-pipelined, it executes one instruction at a time, and has no hazards.

When defining pipelined processors and their specifications, we assume that they do not execute self-modifying code, which allows us to model the Instruction Memory as a read-only memory, separate from a Data Memory if the processor executes load and store instructions. In Figure 2, the Instruction Memory has one read port that takes the PC as address, and produces the four fields of an instruction in the given ISA: RegWrite, a bit indicating whether the instruction will update the Register File; DestReg, destination register identifier; Op, opcode to be used by the ALU; and SrcReg, source register identifier. Alternatively, a read-only instruction memory can be modeled with a collection of uninterpreted functions and uninterpreted predicates, each taking as input the instruction address, and mapping it to a field from the instruction encoding. In the case when some of the above fields do not have a counterpart in the instruction encoding, but are pro-

duced by decoding logic, both models can be viewed as encapsulating the original read-only instruction memory and the decoding logic. To model decoding logic that is separate from the instruction memory, we can use uninterpreted functions and uninterpreted predicates, each mapping a field from the original instruction encoding to a control signal.

```
//-------- 3-stage pipelined processor pipe3.abs  ------------------------
  (bit    //----- Declaration of bit-level signals -----------
     phi1 phi2 phi3 phi4 Flush Flush_bar RegWrite IFD_RegWrite
     IFD_EX_RegWrite EX_WB_RegWrite write_PC write_RegFile_In RegsEqual fwd)
  (term   //----- Declaration of word-level signals ----------
     PC sequentialPC SrcReg DestReg Op IFD_EX_SrcReg IFD_EX_DestReg
     IFD_EX_Op IFD_EX_Data1 Data1 EX_Data1 Result EX_WB_Result EX_WB_DestReg)
  (input phi1 phi2 phi3 phi4 Flush)

  Flush_bar = (not Flush)
  write_PC  = (and phi4 Flush_bar)
  (latch PC       //-------- The Program Counter -------------------------
    (inport  write_PC (sequentialPC))
    (outport phi1     (PC))
  )
  sequentialPC = (PCAdder PC)

  (memory IMem    //-------- The read-only Instruction Memory -----------
    (outport phi2 PC (SrcReg DestReg Op RegWrite))
  )

  write_RegFile_In = (and phi2 EX_WB_RegWrite)
  (memory RegFile //-------- The Register File --------------------------
    (inport  write_RegFile_In EX_WB_DestReg (EX_WB_Result))
    (outport phi3             SrcReg        (Data1))
  )
  IFD_RegWrite = (and RegWrite Flush_bar)

  (latch IFD_EX   //-------- EX Stage Logic ------------------------------
    (inport  phi4 (SrcReg DestReg Op IFD_RegWrite Data1))
    (outport phi1 (IFD_EX_SrcReg IFD_EX_DestReg IFD_EX_Op
                   IFD_EX_RegWrite IFD_EX_Data1))
  )

  RegsEqual = (= IFD_EX_SrcReg EX_WB_DestReg)
  fwd       = (and RegsEqual EX_WB_RegWrite)
  EX_Data1  = (mux fwd EX_WB_Result IFD_EX_Data1)
  Result    = (ALU IFD_EX_Op EX_Data1)

  (latch EX_WB    //-------- WB Stage Logic ------------------------------
    (inport  phi4 (Result IFD_EX_DestReg IFD_EX_RegWrite))
    (outport phi1 (EX_WB_Result EX_WB_DestReg EX_WB_RegWrite))
  )
```

Figure 2. AbsHDL description of the 3-stage pipelined processor pipe3.

Signal `Flush` in Figures 1 and 2, when asserted to 1, is used to disable fetching of instructions and to feed the pipeline with bubbles, allowing partially executed instructions to complete. Then, simulating the pipeline for a sufficient number of clock cycles—as determined by the pipeline depth and possible stalling conditions—will map all partially executed instructions to the architectural state elements (the PC and the Register File in pipe3). The contents of the architectural state elements, with no pending updates in the pipeline, can be directly compared with the contents of the architectural state elements of the specification—see Section 3 for details. In the case of pipe3, which has two pipeline latches and no stalling logic, setting signal `Flush` to 1 and simulating the processor for 2 cycles will complete any instructions that are originally in the pipeline. This mechanism for automatically mapping the state of an implementation processor to its architectural state elements was proposed by Burch and Dill[57]. Note that most processors have a similar signal indicating whether the Instruction Cache provided a valid instruction in the current clock cycle, so that we can achieve the above effect by forcing that signal to the value indicating an invalid instruction. Adding signal `Flush`—to allow completion of partially executed instructions in a pipelined or superscalar processor without fetching new instructions—can be viewed as *design for formal verification*. Signal `Flush`, when set to 1, should invalidate all control bits that indicate updates of architectural state elements.

The phase clocks in an AbsHDL processor description are used to ensure the proper flow of signals in the pipeline stages, as well as to determine the order of memory port operations. In Figure 2, we assume that the phase clocks become high in the order of their numbers. That is, the pipeline latches and the PC are read on `phi1`, then the Register File is written on `phi2`, then the Register File is read on `phi3` (so that the Register File behaves as a write-before-read memory and provides internal forwarding of the result written in the current clock cycle), and finally the pipeline latches and the PC are written on `phi4`, which concludes a clock cycle.

3. Formal Verification Background

The formal verification tool flow is shown in Figure 3. The term-level symbolic simulator TLSim[50] takes an implementation and a specification processor, as well as a simulation-command file indicating how to simulate the implementation and the specification and when to compare their architectural state elements. “Term” in “term-level” comes from the syntax of the logic of Equality with Uninterpreted Functions and Memories (EUFM)[57]—see Figure 5—where terms are used to abstract word-level values. EUFM allows us to mathematically model the behavior of the AbsHDL constructs during symbolic simulation, as well as to define a formula for the correctness of a pipelined processor. In symbolic simulation, the initial state of a processor is represented with variables, while the next state is computed as a function of these variables, based on the description of the processor.

In the sequence of three projects, symbolic simulation is done according to the inductive correctness criterion in Figure 4, which checks the *safety property* of a single-issue pipelined processor—if the processors does something during one step of operation, then it does it correctly. Specifically, if the implementation makes one step starting from an arbitrary initial state Q_{Impl}, then the new implementation state Q'_{Impl} should correspond to the state of the specification after

either 0 or 1 steps, when starting from architectural state Q^0_{Spec} that corresponds to the initial implementation state Q_{Impl}.

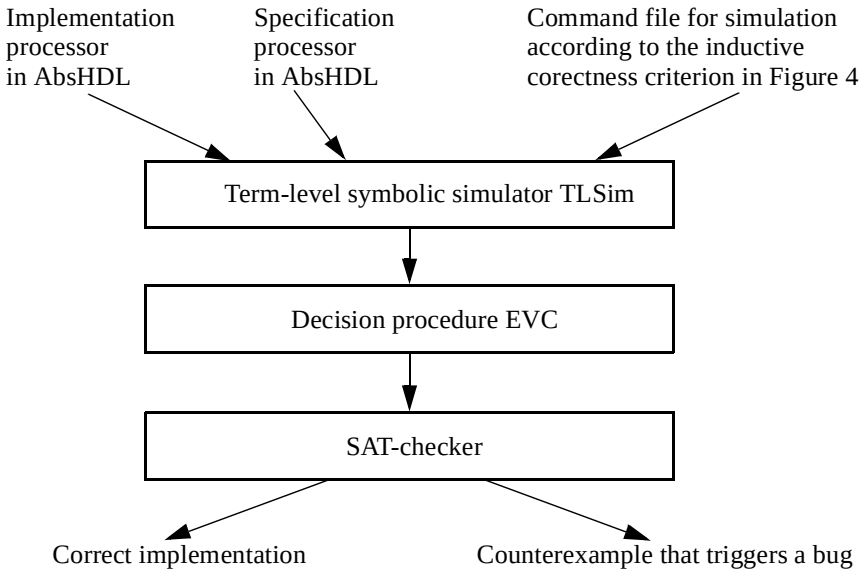


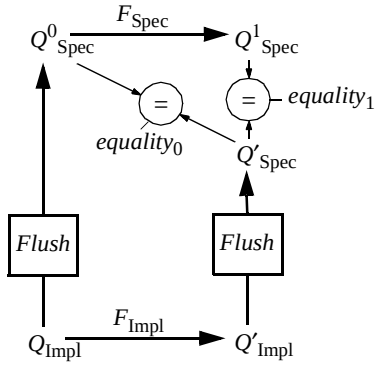
Figure 3. The formal verification tool flow.

Correspondence is determined by flushing the implementation processor[57]—setting signal `Flush` in Figures 1 and 2 to *true* (i.e., 1), and symbolically simulating for a sufficient number of cycles—then comparing the resulting architectural state of the implementation with the state of the specification. In the case of a dual-issue superscalar processor that can fetch and complete up to two instructions per cycle (as in Project 3), the specification is symbolically simulated for two cycles, and the implementation state after one step, Q'_{Impl}, is compared for correspondence to the specification state after 0, or 1, or 2 steps.

In Figure 4, the implementation state after one step, Q'_{Impl}, will correspond to the initial specification state, Q^0_{Spec}, (i.e., condition $equality_0$ will be *true*) when either the implementation does not fetch a new instruction during the one step from Q_{Impl} to Q'_{Impl}, or a new instruction is fetched but is later squashed, e.g., due to a mispredicted branch or a raised exception. Since the example processor pipe3 does not stall or squash instructions, it fetches an instruction on every clock cycle, and each instruction gets completed. Hence, one step of a correct pipe3 should always correspond to one step of the specification.

The commutative diagram in Figure 4 is a correctness proof by induction, since the initial implementation state, Q_{Impl}, is completely arbitrary. If the implementation is correct for all transitions that can be made for one step from an arbitrary initial state, then the implementation will be correct for one step from the next implementation state, Q'_{Impl}, since that state will be a special case of an arbitrary state as used for the initial state, and so on for any number of steps. For some processors, e.g., where the control logic is optimized by using unreachable states as don't-care conditions, we might have to impose a set of *invariant constraints* for the initial implementation

state in order to exclude unreachable states. Then, we need to prove that those constraints will be satisfied in the implementation state after one step, Q'_{Impl}, so that the correctness will hold by induction for that state, and so on for all subsequent states. The reader is referred to Aagaard et al.[58, 59] for a discussion and classification of correctness criteria.



Safety property:

$equality_0 \vee equality_1 = true$

Figure 4. Inductive correctness criterion for single-issue pipelined processors.

In order to account for an arbitrary initial implementation state and for all possible transitions from it, the tool flow employs symbolic simulation. The term-level symbolic simulator TLSim automatically introduces variables for the initial state of memories and latches. TLSim propagates those variables through the processor logic, building symbolic expressions for the values of logic gates, uninterpreted functions, uninterpreted predicates, memories, and latches. The symbolic expressions are defined in the logic of EUFM—see Figure 5.

$$
\begin{aligned}
\textit{term} \quad ::= \quad & \textbf{\textit{ITE}}(\textit{formula}, \textit{term}, \textit{term}) \\
& \mid \textit{uninterpreted-function}(\textit{term}, \ldots, \textit{term}) \\
& \mid \textbf{\textit{read}}(\textit{term}, \textit{term}) \\
& \mid \textbf{\textit{write}}(\textit{term}, \textit{term}, \textit{term}) \\
\textit{formula} \quad ::= \quad & \textit{true} \mid \textit{false} \mid (\textit{term} = \textit{term}) \\
& \mid (\textit{formula} \wedge \textit{formula}) \mid (\textit{formula} \vee \textit{formula}) \mid \neg \textit{formula} \\
& \mid \textit{uninterpreted-predicate}(\textit{term}, \ldots, \textit{term})
\end{aligned}
$$

Figure 5. Syntax of the logic of Equality with Uninterpreted Functions and Memories.

The syntax of EUFM includes terms and formulas. Terms abstract word-level values, such as data, register identifiers, memory addresses, as well as the entire states of memories, and are used to model the datapath of a processor. Formulas represent Boolean signals and are used to model the control path of a processor, as well as to express the correctness condition. A term can be an uninterpreted function (UF) applied on a list of argument terms; a term variable (that can be

viewed as an UF with no arguments); or an *ITE* operator selecting between two argument terms based on a controlling formula, such that *ITE*(*formula*, $term_1$, $term_2$) will evaluate to $term_1$ when *formula = true*, and to $term_2$ when *formula = false*. A formula can be an uninterpreted predicate (UP) applied on a list of argument terms; a propositional variable (an UP with no arguments); or an equality comparison of two terms. Formulas can be negated, conjuncted, or disjuncted. An *ITE* operator of the form *ITE*(f, f_1, f_2), selecting between two argument formulas f_1 and f_2 based on a controlling formula f, can be defined as $f \wedge f_1 \vee \neg f \wedge f_2$. That is, an *ITE* operator is a mathematical representation for a multiplexor. We will refer to both terms and formulas as *expressions*.

UFs and UPs are used to abstract the implementation details of combinational functional units by replacing them with "black boxes" that satisfy no particular properties other than that of *functional consistency*—equal input expressions produce equal output values. Then, it no longer matters whether the original functional unit is an adder or a multiplier, etc., as long as the same UF (or UP) is used to replace it in both the implementation and the specification processor. Note that in this way we will prove a more general problem—that the processor is correct for any functionally consistent implementation of its functional units. However, that more general problem is much easier to prove.

The syntax for terms can be extended to model memories by means of the interpreted functions *read* and *write*. Function *read* takes two argument terms serving as memory state and address, respectively, and returns a term for the data at that address in the given memory. Function *write* takes three argument terms serving as memory state, address, and data, and returns a term for the new memory state. Functions *read* and *write* satisfy the *forwarding property of the memory semantics*: *read*(*write*(*mem*, *waddr*, *wdata*), *raddr*) is equivalent to *ITE*((*raddr = waddr*), *wdata*, *read*(*mem*, *raddr*)), i.e., if this rule is applied recursively, a *read* operation returns the data most recently written to an equal address, or otherwise the initial state of the memory for that address. Versions of *read* and *write* that extend the syntax for formulas can be defined similarly, such that the former returns a formula, while the latter takes a formula as its third argument.

The term-level symbolic simulator TLSim has commands to compare the final implementation state with the specification states after each step of the specification, and to build an EUFM formula for the inductive correctness criterion in Figure 4. That formula is then input to the decision procedure EVC[50] that eliminates all instances of the interpreted functions *read* and *write* by preserving the forwarding property of the memory semantics. Then, uninterpreted functions are eliminated based on syntactic equality, such that exactly the same combinations of term variables that appear as arguments are mapped to the same new term variable representing the output value. Uninterpreted predicates are eliminated similarly, by using new Boolean variables to represent the output values. The result is an equivalent EUFM formula that contains only term variables, equality comparisons between them, Boolean variables, *ITE* operators, and Boolean connectives. Equality comparisons that have the same term variable for both arguments are replaced with the constant *true*. Equality comparisons, where the arguments are two syntactically different term variables i and j, are encoded with a new Boolean variable—an e_{ij} variable[60]. The property of transitivity of equality—if $a = b$ and $b = c$ then $a = c$, where a, b, and c are term variables—has to be enforced with constraints between the e_{ij} variables[61].

The efficiency of EVC is due to a property called Positive Equality[62], stating that the validity of an EUFM formula under a maximally diverse interpretation of the term variables that appear only in positive (not negated) equality comparisons implies the validity of the formula under any interpretation of those term variables. A maximally diverse interpretation means that each such term variable is treated as a distinct constant, which is not equal to any syntactically different term variable. Hence, maximal diversity results in replacing most of the positive equality comparisons—which have two syntactically different arguments—with the constant *false*, instead of with a new e_{ij} Boolean variable, thus dramatically pruning the correctness formula, and resulting in orders of magnitude speedup.

To exploit the property of Positive Equality, a microprocessor designer has to follow some simple modeling restrictions that reduce the number of equality comparisons appearing in both positive and negative polarity in the correctness formula. For example, the equality comparator that determines whether to take a branch-on-equal instruction (see Figure 6.a) appears in positive polarity when controlling the updates of the PC, but in negated polarity when controlling the squashing of subsequent instructions. Abstracting this equality comparator with an uninterpreted predicate (see Figure 6.b) eliminates the negated equality between the data operands that are used to compute the condition whether to take the branch.

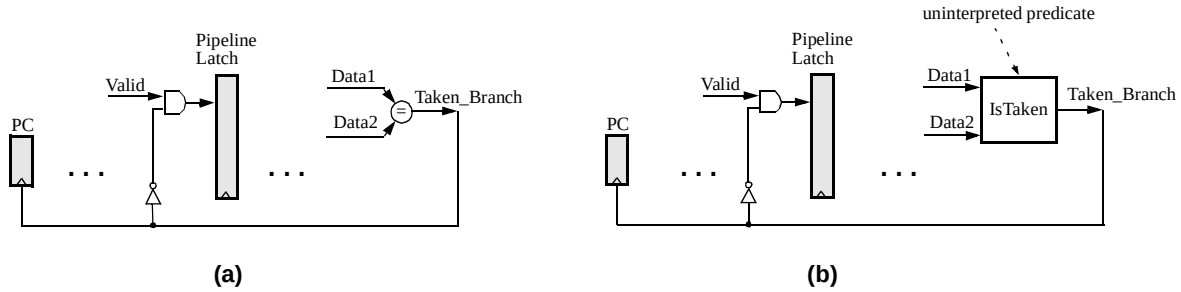


Figure 6. (a) The equality comparator that determines whether to take a branch-on-equal instruction, and driving signal Taken_Branch, appears in positive polarity when controlling the updates of the PC, but in negated polarity when controlling the squashing of subsequent instructions. (b) The equality comparator is abstracted with uninterpreted predicate IsTaken, thus avoiding negated equality between data operands Data1 and Data2.

Similarly, the microprocessor designer has to use the FSM memory model from Figure 7 in order to abstract the Data Memory. Note that the forwarding property of the memory semantics introduces an address equality comparison that controls an *ITE* operator, and so appears in both positive polarity when selecting the then-expression, and in negative polarity when selecting the else-expression. To avoid this dual-polarity equality comparison for the Data Memory, whose addresses are produced by the ALU and also serve as data operands, we use the conservative abstraction shown in Figure 7. There, the original interpreted functions *read* and *write*, that satisfy the forwarding property of the memory semantics, are abstracted with uninterpreted functions *DMem_read* and *DMem_write*, respectively, that do not satisfy this property. The forwarding property is not needed, since all Data Memory accesses complete in program order, with no pending updates by earlier instructions when a later instruction reads data. In contrast, the Register File may not only have pending updates, but may also interact with the load-interlock stalling logic. It

is this interaction that requires the forwarding property[63]. Alternatively, a hybrid memory model[63]—where the forwarding property is preserved only for exactly those pairs of one read and one write address that also determine load-interlock stalling conditions—can be applied automatically. The Data Memory must be abstracted in both the implementation and the specification.

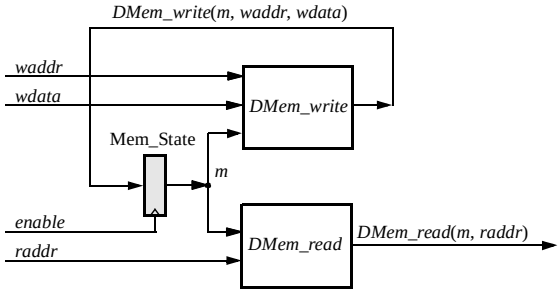


Figure 7. FSM model for conservative abstraction of memories, using uninterpreted functions *DMem_read* and *DMem_write* that do not satisfy the forwarding property of the memory semantics. Term variable *m* abstracts the initial state of the memory.

The decision procedure EVC exploits the property of Positive Equality, and automatically applies other optimizations, translating the EUFM correctness formula to an equivalent Boolean formula, which has to be a tautology for the original EUFM formula to be valid, i.e., for the pipelined implementation to be correct. The Boolean formula can then be checked for being a tautology with any Boolean Satisfiability (SAT) procedure. Any assignment of values to the Boolean variables that makes the Boolean formula *false* will trigger a bug, i.e., is a *counterexample*. In our earlier work[53, 64], we found the SAT-checkers Chaff[54–55] and BerkMin[56] to be most efficient for evaluating the resulting Boolean correctness formulas. The tool flow used in the projects included Chaff. The reader is referred to Zhang and Malik[65] for an overview of recent advances in SAT-checkers.

4. Three Projects on Design and Formal Verification of Pipelined Processors

The projects went through two iterations—one in the summer[42], and another in the fall[43] of 2002. Analysis of frequent student bugs from the summer led to the addition of hints to the fall editions of the projects, as well as of preparatory homework exercises—see Section 5. To divide and conquer the design complexity, and to allow the students to better understand the interaction between various features in a pipelined processor, each of the projects was assigned as a sequence of steps. A step included the extension of a pipelined processor from a previous step or from an earlier project with a new instruction type or a new mechanism. The correct semantics of all the functionality implemented up to and including the new step was defined with a non-pipelined specification processor, given to the students. They were also given the TLSim simulation-command files for the steps of Project 1, but were asked to create their own simulation-command files for Projects 2 and 3. For a more permanent effect from the projects, the students were required to completely describe the processors, as opposed to just fill in the missing parts in provided skeleton code. The following versions were assigned in the fall of 2002.

Project 1: Design and Formal Verification of a Single-Issue Pipelined DLX

Step 1—Implementation of register-register ALU instructions. The students were asked to extend pipe3 to a 5-stage pipelined processor, having the stages of the DLX: instruction fetch (IF), instruction decode (ID), execute (EX), memory (MEM), and write-back (WB). The MEM stage was to be left empty; the Register File to be placed in ID; the ALU and the forwarding logic to be in EX.

Step 2—Implementation of register-immediate ALU instructions. The students had to integrate a multiplexor to select the immediate data value as ALU input.

Step 3—Implementation of store instructions. The FSM model for conservative abstraction of the Data Memory (see Figure 7) was to be added to the MEM stage, left empty so far.

Step 4—Implementation of load instructions. The students were asked to implement the load interlock in ID. They were given the hint that an optimized load-interlock should stall only when a dependent instruction's source operand is actually used.

Step 5—Implementation of (unconditional) jump instructions. The students were required to compute the jump target in EX, and update the PC when the jump is in MEM. The processor was to continue fetching instructions that sequentially follow a jump, but squash them when the jump is in MEM, as a way to lay out the circuitry necessary in Step 6, where the processor was to be biased for branch not taken. The students were given the hint that one of the challenges in this step is the logic for updating the PC: during normal operation, when signal `Flush` is *false*, the PC should be updated as usual; however, during flushing, when signal `Flush` is *true*, the PC should be updated only by instructions that are already in the pipeline, as the purpose of flushing is to complete those instructions without fetching new ones. This mechanism for updating the PC is part of the concept of design for formal verification.

Step 6—Implementation of conditional branch instructions. The students were required to update the PC when a branch is in MEM. The processor was to be biased for branch not taken, i.e., to continue fetching instructions that sequentially follow a branch, and squash them if the branch is taken, or allow them to complete otherwise.

Project 2: Design and Formal Verification of a DLX with Exceptions and Branch Prediction

Step 1—Implementation of ALU exceptions. The goal was to extend the processor from Step 6 of Project 1. The new specification had two additional architectural state elements—an Exception PC (EPC) holding the PC of the last instruction that raised an ALU exception, and a flag IsException that indicates whether an ALU exception was actually raised, i.e., whether the EPC contains valid information. Uninterpreted predicate ALU_Exception was used to check for ALU exceptions by having the same inputs as the ALU and producing a Boolean signal, indicating whether the ALU inputs raised an exception; if so, then the PC of the excepting instruction was stored in the EPC, flag IsException was set to *true*, and the PC was updated with the address of an ALU-exception handler. That address was produced by uninterpreted function ALU_Exception_Handler that takes no arguments, i.e., has a hard-wired value. ALU exceptions can be raised only if control bit allow_ALU_exceptions, produced by the Instruction Memory, is *true*. An instruction that raises an ALU exception does not modify the Register File or the Data Memory.

Step 2—Implementation of a return-from-exception instruction. This instruction updates the PC with the value of the EPC if flag IsException is set, and then clears that flag.

Step 3—Implementation of branch prediction. Since branch prediction is a mechanism that enhances the performance of an implementation processor only, the specification processor was the same as for Step 2 of this project. The branch predictor was abstracted with the FSM in Figure 8, where the present state is stored in latch PresentState; the next state is produced by uninterpreted function Next_FSM_State that depends on the present state only; predictions for the direction (taken or not taken) and the target of a newly fetched branch are generated by uninterpreted predicate PredictedDirection and by uninterpreted function PredictedTarget, respectively, both having as argument the present FSM state. Since the initial FSM state is arbitrary (i.e., there are no restrictions imposed for that state), then the first predicted direction and target will also be arbitrary, and so will be the next FSM state that is produced by applying an arbitrary uninterpreted function to the present state. Note that the only relevant property of the predicted target is whether it is equal or not to the actual target, so that an arbitrary value for the predicted target will account for both cases. Similarly, an arbitrary value for the predicted direction of a branch will account for both possible outcomes. Then, if a processor is correct for an arbitrary prediction of a newly fetched branch or jump, the processor will be correct for any actual implementation of a branch predictor (including one that always predicts incorrectly).

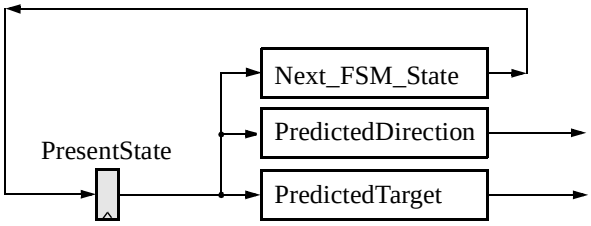


Figure 8. Block diagram for an abstraction of a branch predictor.

The students were required to place the abstraction of the branch predictor in the IF stage, such that if a newly fetched instruction is a branch and the predicted direction is taken, or is a jump (and is so always taken), then the PC should be speculatively updated with the predicted target. The logic for detecting branch/jump mispredictions was to be placed in the MEM stage, since the actual target and direction are computed in the EX stage. The students were given the hint that the implementation processor is simulated for only one cycle of regular operation in Figure 4 and the PC is updated speculatively only when a new instruction is fetched in the IF stage, so that the speculative updates should be disabled during flushing; however, since the purpose of flushing is to complete instructions that are already in the pipeline, PC updates made by the logic for correcting branch/jump mispredictions should be allowed to continue during flushing. This mechanism for updating the PC is part of the concept of design for formal verification.

Project 3: Design and Formal Verification of a Dual-Issue Superscalar DLX

Step 1—Implementation of a base dual-issue DLX. The goal was to extend the processor from Step 4 of Project 1 to a dual-issue superscalar version where the first pipeline can execute all four instruction types, i.e., register-register, register-immediate, store, and load, while the second pipeline can execute only register-register, and register-immediate instructions. The processor was to have in-order issue, in-order execution, and in-order completion. The issue logic was to be based on a shift register—if only the first instruction in ID is issued, then the second instruction in ID moves to the first slot in ID, while only one new instruction is fetched and is placed in the second slot in ID. To simplify the issue logic, if both instructions in ID are issued, the students were given the hint to place the first newly fetched (and older) instruction in the first slot in ID; and to simplify the forwarding and squashing logic, when both instructions in ID can be issued to EX, to allow the older instruction to advance only to the EX stage of the first pipeline.

Step 2—Adding jump and branch instructions to the second pipeline. The students were given the hint that the tricky part is to squash all younger instructions following a jump or a taken branch, and to do that for all possible transitions of the younger instructions.

Project 3 was defined as extension of Project 1, but not of Project 2, in order to introduce one difficulty at a time—superscalar execution—as opposed to a combination of several, including exceptions and branch prediction.

5. Integration of the Three Projects into an Advanced Computer Architecture Course

The three projects were integrated into an advanced computer architecture course that was based on the textbook by Hennessy and Patterson[44]. This discussion is about the Fall'02 version of the course[43]. The slides developed at UC Berkeley[66] were extended with two lectures on formal verification of processors, a lecture on recent research papers, a lecture on reconfigurable computing, and several short topics that were added to existing lectures.

To prepare the students for the lectures on formal verification, several concepts were introduced in 10–15 minute sessions in earlier lectures. The first such session defined symbolic simulation and the *ITE* operator. The second presented the interpreted functions *read* and *write* used for abstracting memories, and the forwarding property of the memory semantics that these functions satisfy. The third illustrated the syntax of AbsHDL with the example processor pipe3 (see Figures 1 and 2) and its non-pipelined specification. The two lectures on formal verification of processors followed next—see Lectures 6 and 7 from Fall'02[43]—and introduced the logic of EUFM (see Figure 5); the inductive correctness criterion (see Figure 4); the formal verification tool flow (see Figure 3); the property of Positive Equality; and the modeling restrictions necessary to exploit that property (see Figures 6 and 7). Another short session was included in the lecture on branch prediction in order to discuss the abstraction of a branch predictor with an FSM (see Figure 8), and the integration of branch prediction in a pipelined processor—a prelude to Project 2. Finally, in the lecture on superscalar execution, special emphasis was made on superscalar issue logic that is based on a shift register, as the one in Project 3.

Several homework exercises were introduced to prepare the students for the three projects. Homework 1 had a problem on symbolic simulation—the students had to manually form the symbolic expressions for small circuits, consisting of 2–3 levels of logic gates and a latch; to prove that two such circuits are equivalent; and to simplify symbolic expressions by accounting for the interaction between the symbolic expression for the enable signal of a latch, and the symbolic expressions for controlling signals of multiplexors that drive the data input of the latch (this part of the exercise was motivated by student bugs from the Summer'02 projects[42]). Homework 2 had a problem on defining the controlling signals for the multiplexors in tree-like forwarding logic. Homework 3 had two relevant problems. The first was to implement the issue logic of a 3-wide superscalar processor with out-of-order execution and out-of-order retirement, capable of executing only ALU and load instructions. The issue logic had to be based on a shift register (to prepare the students for Project 3), such that unissued instructions are shifted to the beginning of the register, and emptied slots are filled with newly fetched instructions.

The second relevant problem in Homework 3 was to design and formally verify a 4-stage pipelined ALU, implemented in the style of the staggered integer ALUs in the Intel Pentium 4[3]. That is, the computation of a result is partitioned into computation of the lower half, and computation of the upper half. The ALU operations are such that the lower half of a result depends on only the lower halves of the operands, while the upper half of a result depends on only the upper halves of the operands and the carry-out from computing the lower half of the same result. Based on these input dependencies, we can compute the two halves in different stages of a pipeline, producing each half with an ALU that is half the original width, and so can be clocked twice as fast. Furthermore, if an ALU result serves as operand of a dependent computation, then the lower half of the result need only be forwarded as a lower operand to the dependent computation, while the upper half of the result need only be forwarded as an upper operand. The benefit from staggered pipelining is that a dependent computation can start execution on each new cycle—i.e., half of the original cycle later, as opposed to one original cycle later—thus allowing sequential chains of dependent computations to be completed twice as fast. The students were provided with a non-pipelined specification. The ALU was modeled with three uninterpreted functions—one abstracting the computation of the lower half of the result; another abstracting the computation of the carry-out from the lower half of the result, based on the same inputs as the first uninterpreted function; and a third abstracting the computation of the upper half of the result, by using the carry-out from the lower half of the result, as well as the upper halves of the two operands. The Register File was implemented with two memories—one storing the lower half of the operands, and another storing the upper half. The first stage of the pipeline was for reading the lower and upper halves of the operands from the corresponding part of the Register File. The second stage—for computing the lower half of the result by using the lower halves of the two operands, and the lower halves of previous results that are still in flight in the pipeline. The third stage—for computing the upper half of the result by using the upper halves of the two operands, the upper halves of previous results that are still in flight in the pipeline, and the carry-out from computing the lower half of the same result in the second stage. The two halves of the result were written back to the corresponding part of the Register File in the fourth stage. The students had to write their own simulation-command file for TLSim. A similar problem can be found in the textbook by Shen and Lipasti[67], except that the students are not given a way to formally verify their implementations of a staggered ALU.

Another homework problem was assigned in the summer of 2002[42]. The students were given an incorrect version of the pipelined DLX from Project 1, and were told that the processor has five bugs that are variants of frequent errors from implementing the same project. The students were asked to inspect the code, identify the bugs, and explain each of them.

In the fall of 2002, Project 1 had to be completed in two weeks, while Projects 2 and 3 in three weeks each. The students were allowed to work in groups of up to three partners.

6. Results and Bug Analysis

Project 1: Design and Formal Verification of a Single-Issue Pipelined DLX

The students worked in 24 groups, 9 with a single student. The minimum time for completing the project was reported by a graduate student working alone—11 hours. The minimum time reported by an undergraduate working alone was 18.5 hours. The average time for completing the project was 22.9 hours. Step 2 was the easiest—16 groups made no bugs, with an average of 0.38 bugs per group. The second easiest was Step 6, where 10 groups made no bugs, with an average of 0.63 bugs per group. This was expected, since both steps involve relatively small changes to the processor from the previous step. Step 5 was the hardest—every group made at least one mistake, and the average number of mistakes per group was 1.86, the highest for all steps. This can be explained with the complexity of adding squashing to a processor with stalling, and correctly handling the case when both mechanisms are triggered.

A pipelined processor for Step 6 was implemented in 400–450 lines of AbsHDL code. Typical verification time for such a processor is less than 1.5 seconds on a 500-MHz Sun Blade 100 workstation: 0.07 seconds for TLSim to symbolically simulate the implementation and the specification; 0.2 seconds for EVC to translate the correctness formula to an equivalent Boolean formula; and 1 second for Chaff to prove the unsatisfiability of the Boolean formula when the implementation is correct. However, Chaff takes less than a second to detect a bug, if one exists.

The students made a total of 44 different bugs when implementing Project 1, not counting AbsHDL syntax errors—TLSim simulates only if the implementation and specification are free of syntax errors, and all signals are either driven or are defined as inputs. In Step 1, 59% of the bugs were made when implementing the forwarding logic. In Step 2, half of all errors were due to incorrect second data input to the MUX selecting the immediate data value as ALU operand—the students used the value of the data operand from the ID_EX latch, instead of the output of the forwarding logic for that operand. In Step 3, half of the bugs were due to incorrect design for formal verification—the students did not invalidate the new control bit MemWrite in the IF stage during flushing. Almost as frequent were bugs due to incorrect data used as input to the Data Memory. In Step 4, the most frequent bug was not forwarding the value loaded from the Data Memory to the ALU in the EX stage. However, the majority of the errors in that step were due to incorrect stalling or incorrect definition of the load interlock. In Step 5, the most frequent bug was related to the concept of design for formal verification—the students did not allow the PC to be updated by jump instructions that are already in the pipeline during flushing. Almost as frequent bug was not accounting for the interaction between stalling and squashing by not squashing the instruction in the IF_ID latch when that latch is stalled due to the load interlock in the ID stage, but there is a

jump instruction in the MEM stage. In Step 6, most mistakes were due to incorrect data source for the uninterpreted predicate that abstracts the branch condition. When defining the load interlock in Step 4, eight groups had an optimized implementation that considers whether the trailing instruction's control bits indicate a pending update of an architectural state element.

Project 2: Design and Formal Verification of a DLX with Exceptions and Branch Prediction

The students worked in 24 groups, 9 with a single student. The minimum time for completing the project was reported by a graduate student working alone—8 hours and 10 minutes. The minimum time reported by an undergraduate student working alone was 12 hours. The average time for completing the project was 27.5 hours. Step 3 took the longest on average—11.3 hours—which can be explained with the complexity of incorporating branch/jump prediction into a pipelined processor, and the many cases that have to be considered when correcting a misprediction.

A pipelined processor for Step 3 was implemented in 650–750 lines of AbsHDL code, depending on the amount of comments. Typical verification time for such a processor is less than 7 seconds: 0.07 seconds for TLSim to symbolically simulate the implementation and the specification; 2 seconds for EVC to translate the correctness formula to an equivalent Boolean formula; and 4.5 seconds for Chaff to prove the unsatisfiability of the Boolean formula when the implementation is correct. However, Chaff takes less than a second to find a satisfying assignment that triggers a bug, if one exists. The required memory was less than 6.6 MB.

The students made a total of 29 different bugs when implementing Project 2. The most frequent bug in Step 1 was not squashing the control bit that allows ALU exceptions for subsequent instructions, when a jump or a taken branch is in the MEM stage. The second most frequent mistake was not including the ALU exception signal as part of the squash condition. In Step 2, the most frequent error was not squashing instructions that follow a return-from-exception. Additionally, many bugs were due to allowing architectural state updates that are disabled in the specification when an instruction is a return-from-exception. In Step 3, half of the mistakes were due to missing cases in the logic for correcting branch/jump mispredictions. A tricky bug was not squashing the bit for the predicted branch direction, when the branch is squashed, so that the logic for correcting mispredictions would still take action to correct such a branch.

Project 3: Design and Formal Verification of a Dual-Issue Superscalar DLX

The students worked in 17 groups, each consisting of 3 partners. The minimum time for completing the project was reported by a group of three undergraduates—12 hours. The average time for completing the project was 29.1 hours. Step 1 took an average of 17.4 hours, which was about 50% longer compared to the average time of 11.8 hours spent on Step 2. This can be explained with the extensive modifications required in Step 1 for adding a second pipeline to the single-issue processor from Step 4 of Project 1, compared to just incorporating jumps and branches in a dual-issue superscalar processor, as done in Step 2. No group completed a step without bugs, indicating the increased difficulty of this project.

A dual-issue superscalar processor from Step 2 was implemented in 900–1,000 lines of AbsHDL code. Typical verification time for such a processor is less than 10 minutes: 0.5 seconds

for TLSim to symbolically simulate the implementation and the specification; 12 seconds for EVC to translate the correctness formula to an equivalent Boolean formula; and 550 seconds for Chaff to prove the unsatisfiability of the Boolean formula when the implementation is correct. However, Chaff takes only a few seconds[53, 64] to find a satisfying assignment if a bug exists. The required memory was 230 MB. The time for SAT-checking with Chaff can be reduced to 6 seconds if the students are taught the concept of controlled flushing[68]—where the user defines a flushing schedule to avoid pipeline stalls, thus simplifying the correctness formula. Alternatively, EVC can be extended with rewriting rules that are applied automatically and achieve the same effect, e.g., as used to formally verify out-of-order superscalar processors[69]. The specifications for Step 1 and 2 were the same as for Steps 4 and 6, respectively, of Project 1.

The students made a total of 20 different bugs when implementing Project 3. The most frequent bug in Step 1 was incorrect priority of the results forwarded to the two ALUs in the EX stage. The rest of the bugs in Step 1 were due to incorrect implementation of the issue logic based on a shift register. In Step 2, the most frequent bug was not squashing an instruction that is not issued and is so moved from the ID stage of the second pipeline to the ID stage of the first pipeline, when a jump or a taken branch is in the MEM stage of the second pipeline. A detailed description of the bugs from the three projects is available separately[70].

Homework Problem on Design and Formal Verification of a Pipelined Staggered ALU

The most common bug made in the design of the pipelined staggered ALU was not forwarding the high result from the fourth (write-back) stage to the high operands in the second stage, where these operands are just transferred to the third stage. Then, when they get in the third stage, and if no forwarding takes place for the next high result that will then be in the fourth stage, these operands will be incorrect and will produce an incorrect high result.

7. Related Work

Traditionally, the principles of pipelined and superscalar execution have been taught by using trace-driven simulation[6–21, 71–74], or FPGA-based hardware emulation[21, 30–34]. Surveys of simulation resources are presented by Wolffe et al.[75, 77], and Yehezkel et al.[76] However, simulator bugs and modeling inaccuracies can lead to significant errors in performance measurements[78–84], and so may result in wrong design decisions. To avoid bugs, Weaver et al.[16] extended their trace-driven simulator with a dynamic checker, which is similar to the checker processor in the DIVA architecture[85], and is used to compare the superscalar simulator results with those produced by a non-pipelined simulator. While such checker can detect bugs triggered by the benchmarks simulated so far, it does not guarantee full correctness, so that bugs may still remain—to be activated by other benchmarks—and affect the performance measurements; it does not identify modeling inaccuracies that lead to imprecise performance measurements; and does not solve the problem of how to prove the correctness of pipelined processors for all execution sequences. The lack of guarantee for complete correctness diminishes the pedagogical power of simulation, when teaching the principles of pipelined, speculative, and superscalar execution.

A DIVA checker[85]—used to dynamically verify the results of an out-of-order execution engine—can itself be a pipelined processor and thus has to be formally verified to ensure the correctness of the entire system. Mneimneh et al.[86] verified the recovery mode of a pipelined DIVA

checker against its ISA. However, in recovery mode, a DIVA checker allows only one instruction in the pipeline at a time, so that mechanisms for avoiding hazards are not triggered, and the checker behaves like a non-pipelined processor, making its verification trivial in that mode. Additionally, an incorrect out-of-order execution engine in the DIVA architecture can degrade the performance of the system, requiring the checker to frequently squash instructions in the out-of-order execution engine and restart the execution after the last correctly completed instruction. Most importantly, single-issue pipelined and dual-issue superscalar processors—which are widely used in embedded systems—may not be possible to integrate with a checker processor due to the stringent silicon area, power-consumption, and cost constraints in that market segment. And because of the explosion of embedded applications, it is more likely that recent university graduates will work on the design of embedded processors than of aggressive out-of-order processors. Hence, the need to educate students with deep understanding of the principles of pipelined, speculative, and superscalar execution; students who can formally verify their designs.

Directions for integrating formal methods into software engineering courses are outlined by Almstrum et al.[87], and by Wing[88]. There are many reports from integrating formal verification into existing software engineering curricula[36–40]. A list of formal verification courses offered at various universities can be found at the Formal Methods Educational Site[35]. However, none of those courses integrates formal methods into an existing computer architecture course. The only such course[19] that the author knows of was taught at Carnegie Mellon University from 1995 till 1998 in a version that included a project on model checking[89] a snoopy cache coherence protocol with SMV[41]. In a different school, the instructor[90] used a formal verification tool to illustrate how to check properties of a snoopy cache coherence protocol and of a bus protocol, when teaching computer architecture, but did not assign related projects or homework exercises.

Another course where the students designed dual-issue superscalar processors, but using VHDL, is reported by Hamblen et al.[33]. However, dual-issue processors were attempted by only two of the nine groups, and one of the designs was found to be buggy. Of the other seven designs, which were single-issue pipelined processors, three were not completely functional.

Functional verification was taught by Ozguner et al.[1] However, the students did not use formal verification tools. Van Campenhout et al.[27, 28] classified 87 student bugs, made in Verilog designs of a 5-stage DLX with branch prediction. When designing the DLX processors in Verilog, those students did not describe the register file and the ALUs—similar to the work presented here—but used library modules. However, their processors were described at a lower level of abstraction; were of lower complexity—did not implement exceptions in addition to branch prediction, or have dual-issue superscalar execution; were not implemented by accounting for the concept of design for formal verification; and were not formally verified—Van Campenhout et al. report that their testing method detected 94% of the student errors.

Lahiri et al.[91] used TLSim and EVC at Motorola, applying the same abstraction techniques as in the projects in order to model exceptions and branch prediction. They formally verified a model of the M•CORE processor[49], detecting three bugs—two in the forwarding logic, and one in the issue logic—as well as several corner cases that were not fully implemented.

When abstracting the functional units and memories (see Section 3), we assume that their bit-level implementations are formally verified separately. Pandey and Bryant[92] combined the method of Symbolic Trajectory Evaluation (STE)[93–97] with symmetry reductions to formally verify large memory arrays at the transistor level. Claesen et al.[98] used theorem proving to formally verify an SRT integer divider. Aagaard and Seger[99] combined model checking and theorem proving to show the correctness of an Intel multiplier. O'Leary et al.[100] formally verified an SRT integer square root circuit. Russinoff[101] proved the correctness of the multiplication, division, and square root algorithms of the AMD K7 processor. Jones et al.[102, 103] combined STE with theorem proving to formally verify an instruction decoder and a floating-point adder/subtracter from Intel processors. Chen et al.[104] used word-level model checking to formally verify an Intel floating-point unit that could add, subtract, multiply, divide, round, and compute square root. Chen and Bryant[105] proved the correctness of floating-point adders and multipliers. Bryant and Chen[106] formally verified large combinational multipliers.

Historically, the inductive correctness condition in Figure 4 dates back to Milner[107] and Hoare[108], who used it to verify programs by manually defining an abstraction function that maps the state of the implementation program to the state of the specification program. Manual abstraction functions were used in initial work on formal verification of processors, as reported by Srivas and Bickford[109], Saxe et al.[110], and Hunt[111]. In other early work, presented by Cohn[112] and Joyce[113], the processors were simple and non-pipelined, so that the abstraction function was computed by just taking the architectural state. Burch and Dill[57] proposed flushing a pipelined processor as a way to automatically compute an abstraction function from the state of the implementation to the state of the specification processor, and were the first to formally verify a pipelined DLX, comparable to that from Project 1. However, the user had to manually provide a case-splitting expression, indicating the conditions when the pipeline will fetch a new instruction in the current cycle and will later complete that instruction. The validity of the EUFM correctness formula was checked by a prototype of the decision procedure SVC[114], which used the case-splitting expression to simplify the correctness formula, and evaluate it twice—once under the condition that the case-splitting expression is *true*, and a second time under the condition that the case-splitting expression is *false*. Jones et al.[115] extended SVC with heuristics that sped up the verification. Burch[68] used the resulting tool, and applied the method developed with Dill[57] to a dual-issue superscalar DLX, comparable to that from Project 3. However, he had to manually define 28 case-splitting expressions, and to decompose the commutative correctness diagram from Figure 4 into 3 commutative diagrams that were easier to verify. That decomposition was subtle enough to warrant publication of its correctness proof as a separate paper[116]. Hosabettu et al.[117, 118] used the theorem-prover PVS[119] and manually defined abstraction functions to formally verify a single-issue pipelined DLX and a dual-issue superscalar DLX, comparable to those from Projects 1 and 3, respectively, but reported 30 days of manual work for each of the processors.

To formally verify 5-stage pipelined DLX or ARM processors (comparable to that from Project 1), experts invested extensive manual work, often reporting long run times[117–118, 120–127]. Even 3-stage pipelines, executing only ALU instructions, took significant manual intervention to formally verify with theorem proving[128–131], or with assume-guarantee reasoning[132–134]. Symbolic Trajectory Evaluation also required extensive manual work to prove the correctness of just a register-immediate `OR` instruction in a bit-level 5-stage ARM processor[135]. Other researchers had to limit the data values to 4 bits, the register file to 1 register, and the ISA to 16 instructions, to

symbolically verify a bit-level pipelined processor[136]. Various symbolic tools ran for a long time when formally verifying a pipelined DLX[137–140], or ran out of memory[141]. Custom-tailored, manually defined rewriting rules were used to formally verify a 5-stage DLX[142, 143], and similar 4-stage processors[144–147], but would require modifications to work on designs described in different coding style, and significant extensions to scale for dual-issue superscalar processors. Other researchers proved only few properties of a pipelined DLX[148, 149], or did not present completeness argument[150, 151]—that the properties proved ensure correctness under all possible scenarios.

Note that by formally verifying a processor, we prove the logical correctness of the design. However, fabrication defects may still lead to bugs in specific chips, and can only be detected by testing methods[152–162].

To summarize, testing methods do not guarantee correctness, as well as take significant time to apply, which diminishes their pedagogical power when teaching the principles of pipelined, speculative, and superscalar execution. Furthermore, previous formal verification approaches did not scale, or required extensive manual effort and level of expertise that made them impossible to integrate in existing advanced computer architecture courses, where the students are not required to have prior knowledge of formal methods, and complete projects on design and formal verification of pipelined and superscalar processors in addition to other assignments. In contrast, the tool flow—TLSim, EVC, and efficient SAT-checker—automatically and quickly proves the correctness of processors from Projects 1–3. The user only defines the command sequence for symbolic simulation with TLSim—based on the inductive correctness condition in Figure 4—using commands that are similar to those of binary simulators.

8. Conclusions and Future Work

The experience presented in this paper indicates that it is possible to integrate formal verification into an existing advanced computer architecture course, taught to both undergraduate and graduate students with no prior knowledge of formal methods. The high-level hardware description language AbsHDL allowed the students to quickly design the processors in a sequence of three projects: 1) a single-issue pipelined DLX; 2) an extension with exceptions and branch prediction, where the branch predictor was abstracted, but the mechanism for correcting branch mispredictions was completely implemented; and 3) a dual-issue superscalar DLX. The highly efficient decision procedure EVC[50], combined with a state-of-the-art SAT-checker[54–56], made possible the fast formal verification of the above processors—requiring 1.5 seconds to formally verify a single-issue pipelined DLX, 7 seconds to formally verify an extension with exceptions and branch prediction, and 10 minutes to formally verify a dual-issue superscalar version. In the case of a buggy pipelined or superscalar processor, the tool flow takes significantly less time to find a condition that triggers the bug, compared to the time to prove the correctness of a bug-free version of the same design.

The author is not aware of another computer architecture course where the students have completed as many processor design projects of the above complexity, in addition to homeworks, paper summaries, and paper comparisons. This illustrates the increased productivity possible with a high-level hardware description language such as AbsHDL. Most importantly, the students had

no prior knowledge of formal methods, and were able to formally verify their implementations of the three projects. A related homework problem was to design and formally verify a staggered ALU, pipelined in the style of the integer ALUs in the Intel Pentium 4[3].

Future work will include the development of visualization tools to help quickly analyze counterexamples from incorrect processors. A sophisticated version of such tools can have the capability to propose patches of logic to replace existing pieces of circuitry in a way that will correct a detected bug. A second direction for future work will be to automate the generation of trace-driven simulators from a formally verified high-level description of a pipelined or superscalar processor; this will allow the students to run simulations and measure the performance of their implementations. A third direction will be to automate the translation of a formally verified high-level processor description to synthesizable Verilog or VHDL, and thus create a path to existing EDA tools for synthesis and rapid prototyping.

Future extensions of the projects will include design and formal verification of VLIW and superpipelined processors, possibly having multiple specialized execution pipelines as in the Intel XScale[163] and the Intel Itanium[164], as well as implementing mechanisms such as scoreboarding[44, 163], predicated execution[52, 164], and advanced loads[52, 164]. Other variations can include pipelined processors with load-value and data-value prediction[67], or with caches that have way prediction[44]. The formal verification of processors with scoreboarding will require the students to impose and check invariant constraints for the pipelines. Formal verification of liveness[165], and of implementations with non-determinism[51] can also be included.

The integration of formal verification in computer architecture courses will result in future microarchitects with deeper understanding of the principles of pipelined, speculative, and superscalar execution; microarchitects who are thus more productive, and capable of delivering correct new processors under aggressive time-to-market schedules.

References

1. F. Ozguner, D. Marhefka, J. DeGroat, B. Wile, J. Stofer, and L. Hanrahan, "Teaching Future Verification Engineers: The Forgotten Side of Logic Design," *38th Design Automation Conference (DAC '01)*, June 2001.
2. B. Bentley, "Validating the Intel® Pentium® 4 Microprocessor," *38th Design Automation Conference (DAC '01)*, June 2001, pp. 244–248.
3. G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, P. Roussel, "The Microarchitecture of the Pentium® 4 Processor," Intel Technology Journal, 1st Quarter, 2001.
4. D. Bhandarkar, and J. Ding, "Performance Characterization of the Pentium® Pro Processor," *3rd International Symposium on High Performance Computer Architecture (HPCA)*, February 1997, pp. 288–297.
5. D. Papworth, "Tuning the Pentium Pro Microarchitecture," IEEE Micro, Vol. 16, No. 2 (April 1996), pp. 8–15.
6. D. Burger, and T. Austin, "The SimpleScalar Tool Set, Version 2.0," University of Wisconsin, Computer Sciences Technical Report 1342, June 1997.
7. T. Diep, "VMW: A Visualization-Based Microarchitecture Workbench," Ph.D. Thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, June 1995.
8. DLXview, http://yara.ecn.purdue.edu/~teamaaa/dlxview/.
9. J. Larus, "SPIM: A MIPS R2000/R3000 Simulator," http://www.cs.wisc.edu/~larus/spim.html.
10. P. Lopez, DLXV, http://dlxv.disca.upv.es/tools/dlxv.html.

11. Mic-1 Simulator, http://www.ontko.com/mic1/.
12. Microarchitecture Simulator 1.1, http://www.dslextreme.com/users/fabrizioo/msim.html.
13. C. Moura, "*SuperDLX—A Generic Superscalar Simulator*," M.S. Thesis, Advanced Compilers, Architecture and Parallel Systems Group, McGill University, May 1993.
14. V.S. Pai, P. Ranganathan, and S.V. Adve, "RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors," *Workshop on Computer Architecture Education (WCAE '97)*, February 1997.
15. SimOS, http://simos.stanford.edu.
16. C.T. Weaver, E. Larson, and T. Austin, "Effective Support of Simulation in Computer Architecture Instruction," *Workshop on Computer Architecture Education (WCAE '02)*, May 2002, pp. 48–55.
17. WinDLX, ftp://ftp.mkp.com/pub/dlx/.
18. M. Wolff, and L. Wills, "SATSim: A Superscalar Architecture Trace Simulator Using Interactive Animation," *Workshop on Computer Architecture Education (WCAE '00)*, June 2000.
19. R.E. Bryant, and T.C. Mowry, Carnegie Mellon University, *CS 740: Basic Computer Systems*, Fall 1998, http://www-2.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15740-f98/www/home.html.
20. M. Brorsson, "MipsIt—A Simulation and Development Environment Using Animation for Computer Architecture Education," *Workshop on Computer Architecture Education (WCAE '02)*, May 2002, pp. 65–72.
21. T. Tateoka, M. Suzuki, K. Kono, Y. Maeda, and K. Abe, "An Integrated Laboratory for Computer Architecture and Networking," *Workshop on Computer Architecture Education (WCAE '02)*, May 2002.
22. D.E. Thomas, and P.R. Moorby, *The Verilog® Hardware Description Language*, 4th edition, Kluwer Academic Publishers, Boston/Dordrecht/London, 1998.
23. P.J. Ashenden, *The Designer's Guide to VHDL*, Morgan Kaufmann Publishers, San Francisco, 1996.
24. P.J. Ashenden, *The Student's Guide to VHDL*, Morgan Kaufmann Publishers, San Francisco, 1998.
25. T.C. Huang, R.W. Melton, P.R. Bingham, C.O. Alford, and F. Ghannadian, "The Teaching of VHDL in Computer Architecture," *International Conference on Microelectronics Systems Education (MSE '97)*, July 1997.
26. SuperScalar DLX, http://www.rs.e-technik.tu-darmstadt.de/TUD/res/dlxdocu/SuperscalarDLX.html.
27. D. Van Campenhout, H. Al-Asaad, J.P. Hayes, T. Mudge, R.B. Brown, "High-Level Design Verification of Microprocessors via Error Modeling," ACM Transactions on Design Automation of Electronic Systems, Vol. 3, No. 4 (October 1998), pp. 581–599.
28. D. Van Campenhout, T. Mudge, J.P. Hayes, "Collection and Analysis of Microprocessor Design Errors," IEEE Design & Test of Computers, Vol. 17, No. 4 (October–December 2000), pp. 51–60.
29. E. Miller, and J. Squire, "esim: A Structural Design Language and Simulator for Computer Architecture Education," *Workshop on Computer Architecture Education (WCAE '00)*, June 2000.
30. G. Brown, and N. Vrana, "A Computer Architecture Laboratory Course Using Programmable Logic," IEEE Transactions on Education, Vol. 38, No. 2 (May 1995), pp. 118–125.
31. N.L.V. Calazans, F.G. Moraes, and C.A.M. Marcon, "Teaching Computer Organization and Architecture with Hands-on Experience," *32nd ASEE/IEEE Frontiers in Education Conference (FIE '02)*, November 2002.
32. N.L.V. Calazans, and F.G. Moraes, "Integrating the Teaching of Computer Organization and Architecture with Digital Hardware Design Early in Undergraduate Courses," IEEE Transactions on Education, Vol. 44, No. 2 (May 2001), pp. 109–119.
33. J.O. Hamblen, H.L. Owen, S. Yalamanchili, and B. Dao, "An Undergraduate Computer Engineering Rapid Systems Prototyping Design Laboratory," IEEE Transactions on Education, Vol. 42. No. 1 (February 1999).
34. S.K. Reinhardt, "Integrating Hardware and Software Concepts in a Microprocessor-Based System Design Lab," *Workshop on Computer Architecture Education (WCAE '00)*, June 2000.
35. Formal Methods Educational Site, http://www.cs.indiana.edu/formal-methods-education/Courses/.
36. D. Garlan, "Integrating Formal Methods into a Professional Master of Software Engineering Program," *8th Z User Meeting (ZUM '94)*, June 1994.
37. J.P. Gibson, and D. Méry, "Teaching Formal Methods: Lessons to Learn," *2nd Irish Workshop on Formal Methods (IWFM '98)*, September 1998.
38. J.P. Gibson, "Formal Requirements Engineering: Learning From the Students," *Australian Software Engineering Conference*, D. Grant, *ed*., 2000, pp. 171–181.
39. A.E.K. Sobel, "Final Results of Incorporating an Operational Formal Method Into a Software Engineering Curriculum," *29th ASEE/IEEE Frontiers in Education Conference (FIE '99)*, November 1999.

40. G. Tremblay, "An Undergraduate Course in Formal Methods: Description Is Our Business," *SIGCSE Technical Symposiu on Computer Science Education*, 1998, pp. 166–170.
41. K.L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, Boston/Dordrecht/London, 1993.
42. M.N. Velev, Georgia Institute of Technology, *ECE 4100, Advanced Computer Architecture*, Summer 2002, http://users.ece.gatech.edu/~mvelev/summer02/ece4100/.
43. M.N. Velev, Georgia Institute of Technology, *ECE 4100/6100, Advanced Computer Architecture*, Fall 2002, http://users.ece.gatech.edu/~mvelev/fall02/ece6100/.
44. J.L. Hennessy, and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd edition, Morgan Kaufmann Publishers, San Francisco, 2002.
45. Intel Corporation, Intel Pentium® Processor, http://developer.intel.com/design/pentium.
46. Digital Equipment Corporation, *Digital Semiconductor Alpha 21064 and 21064A Microprocessors: Hardware Reference Manual*, June 1996.
47. IDT Corporation, *IDT 79RC5000™ RISC Microprocessor Reference Manual*, September 2000.
48. IBM Corporation, PowerPC 440 Embedded Core, http://www-3.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_440_Embedded_Core.
49. Motorola Corporation, Semiconductor Products, http://e-www.motorola.com.
50. M.N. Velev, and R.E. Bryant, "EVC: A Validity Checker for the Logic of Equality with Uninterpreted Functions and Memories, Exploiting Positive Equality and Conservative Transformations," *Computer-Aided Verification (CAV '01)*, G. Berry, H. Comon, and A. Finkel, *eds.*, LNCS 2102, Springer-Verlag, July 2001.
51. M.N. Velev, and R.E. Bryant, "Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction," *37th Design Automation Conference (DAC '00)*, June 2000.
52. M.N. Velev, "Formal Verification of VLIW Microprocessors with Speculative Execution," *Computer-Aided Verification (CAV '00)*, E.A. Emerson, and A.P. Sistla, *eds.*, LNCS 1855, Springer-Verlag, July 2000.
53. M.N. Velev, and R.E. Bryant, "Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors," *38th Design Automation Conference (DAC '01)*, June 2001.
54. M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *38th Design Automation Conference (DAC '01)*, June 2001, pp. 530–535.
55. L. Zhang, C.F. Madigan, M.W. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," *International Conference on Computer-Aided Design (ICCAD '01)*, November 2001.
56. E. Goldberg, and Y. Novikov, "BerkMin: A Fast and Robust Sat-Solver," *Design, Automation, and Test in Europe (DATE '02)*, March 2002, pp. 142–149.
57. 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.
58. M.D. Aagaard, N.A. Day, and M. Lou, "Relating Multi-Step and Single-Step Microprocessor Correctness Statements," *Formal Methods in Computer-Aided Design (FMCAD '02)*, M.D. Aagaard, and J.W. O'Leary, *eds.*, LNCS 2517, Springer-Verlag, November 2002, pp. 123–141.
59. M.D. Aagaard, B. Cook, N.A. Day, and R.B. Jones, "A Framework for Superscalar Microprocessor Correctness Statements," to appear in Software Tools for Technology Transfer (STTT), 2002.
60. A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal, "BDD Based Procedures for a Theory of Equality with Uninterpreted Functions," *Computer-Aided Verification (CAV '98)*, A.J. Hu and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 244–255.
61. R.E. Bryant, and M.N. Velev, "Boolean Satisfiability with Transitivity Constraints," ACM Transactions on Computational Logic (TOCL), Volume 3, Number 4 (October 2002), pp. 604–627.
62. R.E. Bryant, S. German, and M.N. Velev, "Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic," ACM Transactions on Computational Logic (TOCL), Volume 2, Number 1 (January 2001), pp. 93–134.
63. M.N. Velev, "Automatic Abstraction of Memories in the Formal Verification of Superscalar Microprocessors," *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, T. Margaria, and W. Yi, *eds.*, LNCS 2031, Springer-Verlag, April 2001, pp. 252–267.
64. M.N. Velev, and R.E. Bryant, "Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors," Journal of Symbolic Computation (JSC), 2003.
65. L. Zhang, and S. Malik, "The Quest for Efficient Boolean Satisfiability Solvers," *Computer-Aided Verification (CAV '02)*, E. Brinksma, and K.G. Larsen, *eds.*, LNCS 2404, Springer-Verlag, July 2002, pp. 17–36.

66. D.A. Patterson, University of California at Berkeley, *CS 252, Graduate Computer Architecture*, http://www.cs.berkeley.edu/~pattrsn/252S01/index.html.
67. J.P. Shen, and M. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, beta edition, McGraw-Hill, July 2002.
68. J.R. Burch, "Techniques for Verifying Superscalar Microprocessors," *33rd Design Automation Conference (DAC '96)*, June 1996, pp. 552–557.
69. M.N. Velev, "Using Rewriting Rules and Positive Equality to Formally Verify Wide-Issue Out-Of-Order Microprocessors with a Reorder Buffer," *Design, Automation and Test in Europe (DATE '02)*, March 2002.
70. M.N. Velev, "Collection of High-Level Microprocessor Bugs from Formal Verification of Pipelined and Superscalar Designs," submitted for publication.
71. L.B. Hostetler, and Brian Mirtich, "DLXsim—A Simulator for DLX."
72. N. Manjikian, "Enhancements and Applications of the SimpleScalar Simulator for Undergraduate and Graduate Computer Architecture Education," *Workshop on Computer Architecture Education (WCAE '00)*, June 2000.
73. MipSim, http://mouse.vlsivie.tuwien.ac.at/lehre/rechnerarchitekturen/download/Simulatoren/MIPSim.txt.
74. A.S. Tanenbaum, *Structured Computer Organization*, 4th edition, Prentice Hall, 1999.
75. G. Wolffe, W. Yurcik, H. Osborne, and M. Holliday, "Teaching Computer Organization/Architecture With Limited Resources Using Simulators," *33rd Technical Symposium of Computer Science Education (SIGCSE '02)*, February–March 2002.
76. C. Yehezkel, W. Yurcik, and M. Pearson, "Teaching Computer Architecture with a Computer-Aided Learning Environment: State-of-the-Art Simulators," *International Conference on Simulation and Multimedia in Engineering Education (ICSEE '01)*, January 2001.
77. W. Yurcik, G. Wolffe, and M. Holliday, "A Survey of Simulators Used in Computer Organization/Architecture Courses," *Summer Computer Simulation Conference (SCSC '01)*, July 2001.
78. B. Black, A.S. Huang, M.H. Lipasti, and J.P. Shen, "Can Trace-Driven Simulators Accurately Predict Superscalar Performance?", *International Conference on Computer Design (ICCD '96)*, October 1996.
79. B. Black, and J.P. Shen, "Calibration of Microprocessor Performance Models," IEEE Computer, Vol. 31, No. 5 (May 1998), pp. 59–65.
80. P. Bose, and T.M. Conte, "Performance Analysis and Its Impact on Design," IEEE Computer, Vol. 31, No. 5 (May 1998), pp. 41–49.
81. H.W. Cain, K.M. Lepak, B.A. Schwartz, and M.H. Lipasti, "Precise and Accurate Processor Simulation," *Workshop on Computer Architecture Evaluation Using Commercial Workloads*, in conjunction with HPCA, February, 2002.
82. R. Desikan, D. Burger, and S.W. Keckler, "Measuring Experimental Error in Microprocessor Simulation," *28th International Symposium on Computer Architecture (ISCA)*, July 2001, pp. 266–277.
83. R. Desikan, D. Burger, S.W. Keckler, L. Cruz, F. Latorre, A. González, M. Valero, "Errata on Measuring Experimrntal Error in Microprocessor Smulation," ACM SIGARCH Computer Architecture News, Vol. 30, No. 1 (March 2002), pp 2–4.
84. J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich, "Flash vs. (Simulated) Flash: Closing the Simulation Loop," *9th International Symposium on Architectural Support for Programming Languages and Operating Systems*, November 2000, pp. 49–58.
85. T. Austin, "DIVA: A Dynamic Approach to Microprocessor Verification," Journal of Instruction-Level Parallelism (JILP), Vol. 2, June 2000.
86. M. Mneimneh, F. Aloul, C. Weaver, S. Chatterjee, K. Sakallah, and T. Austin, "Scalable Hybrid Verification of Complex Microprocessors," *38th Design Automation Conference (DAC '01)*, June 2001, pp. 41–46.
87. V.L. Almstrum, C.N. Dean, D. Goelman, T.B. Hilburn, and J. Smith, "Support for Teaching Formal Methods: Report of the ITiCSE 2000 Working Group on Formal Methods Education," September 2000. http://www.cs.utexas.edu/users/csed/FM/work/final-v5-7.pdf.
88. J.M. Wing, "Weaving Formal Methods into the Undergraduate Computer Science Curriculum," *8th International Conference on Algebraic Methodology and Software Technology (AMAST '00)*, May 2000.
89. E.M. Clarke, Jr., O. Grumberg, and D.A. Peled, *Model Checking*, MIT Press, Cambridge, 2000.
90. L. Ivanov, "Integrating Formal Verification into Computer Organization and Architecture Courses," *17th Eastern Small College Computing Conference (ESCCC '01)*, October 2001.

91. S. Lahiri, C. Pixley, and K. Albin, "Experience with Term Level Modeling and Verification of the M•CORE™ Microprocessor Core," *6th Annual IEEE International Workshop on High Level Design, Validation and Test (HLDVT '01)*, November 2001, pp. 109–114.
92. M. Pandey, and R.E. Bryant, "Exploiting Symmetry When Verifying Transistor-Level Circuits by Symbolic Trajectory Evaluation," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, No. 7 (July 1999), pp. 918–935.
93. D.L. Beatty, "A Methodology for Formal Hardware Verification with Application to Microprocessors," Ph.D. Thesis, Technical Report CMU-CS-93-190, School of Computer Science, Carnegie Mellon University, 1993.
94. Intel Corporation, *Partial Bibliography of STE Related Research*, http://intel.com/research/scl/library STE_Bibliography.pdf.
95. A. Jain, "Formal Hardware Verification by Symbolic Trajectory Evaluation," Ph.D. Thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, August 1997.
96. K.L. Nelson, "A Methodology for Formal Hardware Verification Based on Symbolic Trajectory Evaluation," Ph.D. Thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1999.
97. C.-J.H. Seger, and R.E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," Formal Methods in System Design, Vol. 6, No. 2 (March 1995), pp. 147–190.
98. L. Claesen, D. Verkest, and H. De Man, "A Proof of the Non-Restoring Division Algorithm and Its Implementation on an ALU," Formal Methods in System Design, Vol. 5 (1994), pp. 5–31.
99. M.D. Aagaard, and C.-J.H. Seger, "The Formal Verification of a Pipelined Double-Precision IEEE Floating-Point Multiplier," *International Conference on Computer-Aided Design (ICCAD '95)*, November 1995.
100. J.W. O'Leary, M.E. Leeser, J. Hickey, and M.D. Aagaard, "Non-Restoring Integer Square Root: A Case Study in Design by Principled Optimization," LNCS 901, 1995.
101. D.M. Russinoff, "A Mechanically Checked Proof of IEEE Compliance of the Floating Point Multiplication, Division and Square Root Algorithms of the AMD K7 Processor," LMS Journal of Computation and Mathematics, No. 1 (1998), pp. 148–200.
102. R.B. Jones, J.W. O'Leary, C.-J.H. Seger, M.D. Aagaard, and T.F. Melham, "Practical Formal Verification in Microprocessor Design," IEEE Design & Test of Computers, Vol. 18, No. 4 (July–August 2001), pp. 16–25.
103. R.B. Jones, *Symbolic Simulation Methods for Industrial Formal Verification*, Kluwer Academic Publishers, Boston/Dordrecht/London, 2002.
104. Y.-A. Chen, E. Clark, P.-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J. O'Leary, and X. Zhao, "Verification of All Circuits in a Floating-Point Unit Using Word-Level Model Checking," *Formal Methods in Computer-Aided Design (FMCAD '96)*, M. Srivas, and A. Camilleri, *eds.*, LNCS 1166, Springer-Verlag, November 1996.
105. Y.-A. Chen, and R.E. Bryant, "An Efficient Graph Representation for Arithmetic Circuit Verification," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 20, No. 12 (December 2001), pp. 1442–1454.
106. R.E. Bryant, and Y.-A. Chen, "Verification of Arithmetic Circuits Using Binary Moment Diagrams," Software Tools for Technology Transfer (STTT), Vol. 3, No. 2 (May 2001), pp. 137–155.
107. R. Milner, "An Algebraic Definition of Simulation Between Programs," *2nd International Joint Conference on Artificial Intelligence*, The British Computer Society, 1971, pp. 481–489.
108. C.A.R. Hoare, "Proof of Correctness of Data Representations," Acta Informatica, Vol. 1, 1972, pp. 271–281.
109. M. Srivas, and M. Bickford, "Formal Verification of a Pipelined Microprocessor," IEEE Software, Vol. 7, No. 5 (September–October 1990), pp. 52–64.
110. J.B. Saxe, S.J. Garland, J.V. Guttag, and J.J. Horning, "Using Transformations and Verification in Circuit Design," Formal Methods in System Design, Vol. 3, No. 3 (December 1993), pp. 181–209.
111. W.A. Hunt, Jr., *FM8501: A Verified Microprocessor*, LNAI 795, Springer-Verlag, 1994.
112. A.J. Cohn, "The Notion of Proof in Hardware Verification," *Journal of Automated Reasoning*, Vol. 5 (1989).
113. J. Joyce, "Multi-Level Verification of Microprocessor-Based Systems," Ph.D. Thesis, Computer Laboratory, Cambridge University, 1989.
114. Stanford Validity Checker (SVC), http://sprout.Stanford.EDU/SVC.
115. R.B. Jones, D.L. Dill, and J.R. Burch, "Efficient Validity Checking for Processor Verification," *International Conference on Computer-Aided Design (ICCAD '95)*, November 1995, pp. 2–6.
116. P.J. Windley, and J.R. Burch, "Mechanically Checking a Lemma Used in an Automatic Verification Tool," *Formal Methods in Computer-Aided Design (FMCAD '96)*, M. Srivas, and A. Camilleri, *eds.*, LNCS 1166, Springer-Verlag, November 1996, pp. 362–376.

117. R. Hosabettu, M. Srivas, and G. Gopalakrishnan, "Decomposing the Proof of Correctness of Pipelined Microprocessors," *Computer-Aided Verification (CAV '98)*, A.J. Hu, and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 122–134.
118. R.M. Hosabettu, "Systematic Verification of Pipelined Microprocessors," Ph.D. Thesis, Department of Computer Science, University of Utah, August 2000.
119. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas, "A Tutorial Introduction to PVS," *Workshop on Industrial-Strength Formal Specification Techniques*, April 1995.
120. E. Börger, and S. Mazzanti, "A Practical Method for Rigourously Controllable Hardware Design," *10th International Conference of Z Users (ZUM '97)*, J. Bowen, M. Hinchey, and D. Till, *eds.*, LNCS 1212, Springer-Verlag, April 1997, pp. 151–187.
121. D. Cyrluk, "Inverting the Abstraction Mapping: A Methodology for Hardware Verification," *Formal Methods in Computer-Aided Design (FMCAD '96)*, M. Srivas and A. Camilleri, *eds.*, LNCS 1166, Springer-Verlag, November 1996, pp. 172–186.
122. J.K. Huggins, and D. Van 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.
123. C. Jacobi, and D. Kröning, "Proving the Correctness of a Complete Microprocessor," *30. Jahrestagung der Gesellshaft für Informatik*, Springer-Verlag, 2000.
124. D. Kröning, and W.J. Paul, "Automated Pipeline Design," *38th Design Automation Conference (DAC '01)*, June 2001, pp. 810–815.
125. S.M. Müller, and W.J. Paul, *Computer Architecture: Complexity and Correctness*, Springer-Verlag, 2000.
126. S. Tahar and R. Kumar, "A Practical Methodology for the Formal Verification of RISC Processors," Formal Methods in Systems Design, Vol. 13, No. 2 (September 1998), Kluwer Academic Publishers, pp. 159–225.
127. P.J. Windley, "Verifying Pipelined Microprocessors," *Conference on Hardware Description Languages (CHDL '95)*, August 1995, pp. 503–511.
128. P. Manolios, "Correctness of Pipelined Machines," *Formal Methods in Computer-Aided Design (FMCAD '00)*, W.A. Hunt, Jr., and S.D. Johnson, *eds.*, LNCS 1954, Springer-Verlag, November 2000, pp. 161–178.
129. P. Manolios, "Mechanical Verification of Reactive Systems," Ph.D. Thesis, Department of Computer Sciences, University of Texas at Austin, August 2001.
130. J. Sawada, "Formal Verification of an Advanced Pipelined Machine," Ph.D. Thesis, Department of Computer Sciences, University of Texas at Austin, December 1999.
131. J. Sawada, "Verification of a Simple Pipelined Machine Model," in *Computer-Aided Reasoning: ACL2 Case Studies*, M. Kaufmann, P. Manolios, and J.S. Moore, *eds.*, Kluwer Academic Publishers, Boston/Dordrecht/ London, 2000, pp. 137–150.
132. T.A. Henzinger, S. Qadeer, and S.K. Rajamani, "You Assume, We Guarantee: Methodology and Case Studies," *Computer-Aided Verification (CAV '98)*, A.J. Hu, and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 440–451.
133. T.A. Henzinger, S. Qadeer, and S.K. Rajamani, "Decomposing Refinement Proofs Using Assume-Guarantee Reasoning," *International Conference on Computer-Aided Design (ICCAD '00)*, November 2000.
134. S. Qadeer, "Algorithms and Methodology for Scalable Model Checking**,"** Ph.D. Thesis, Electrical Engineering and Computer Sciences Department, University of California at Berkeley, Fall 1999.
135. V.A. Patankar, A. Jain, and R.E. Bryant, "Formal Verification of an ARM Processor," *12th International Conference on VLSI Design*, January 1999, pp. 282–287.
136. V. Bhagwati, and S. Devadas, "Automatic Verification of Pipelined Microprocessors," *31st Design Automation Conference (DAC '94)*, June 1994, pp. 603–608.
137. H. Hinrichsen, "Formally Correct Construction of a Pipelined DLX Architecture," Technical Report 98-5-1, Darmstadt University of Technology, May 1998.
138. H. Hinrichsen, H. Eveking, and G. Ritter, "Formal Synthesis for Pipeline Design," *2nd International Conference on Discrete Mathematics and Theoretical Computer Science (DMTCS '99) and the 5th Australasian Theory Symposium (CATS '99)*, C. S. Calude, and M. J. Dinneen, *eds.*, Australian Computer Science Communications, Vol. 21, No. 3, Springer-Verlag, 1999.
139. G. Ritter, H. Eveking, and H. Hinrichsen, "Formal Verification of Designs with Complex Control by Symbolic Simulation," *Correct Hardware Design and Verification Methods (CHARME '99)*, L. Pierre and T. Kropf, *eds.*, LNCS 1703, Springer-Verlag, September 1999, pp. 234–249.

140. G. Ritter, “Formal Sequential Equivalence Checking of Digital Systems by Symbolic Simulation,” Ph.D. Thesis, Department of Electrical and Computer Engineering, Darmstadt University of Technology, 2001.
141. A.J. Isles, R. Hojati, and R.K. Brayton, “Computing Reachable Control States of Systems Modeled with Uninterpreted Functions and Infinite Memory,” *Computer-Aided Verification (CAV ’98)*, A.J. Hu, and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 256–267.
142. J. Levitt, and K. Olukotun, “Verifying Correct Pipeline Implementation for Microprocessors,” *International Conference on Computer-Aided Design (ICCAD ’97)*, November 1997, pp. 162–169.
143. J.R. Levitt. "Formal Verification Techniques for Digital Systems," Ph.D. Thesis, Department of Electrical Engineering, Stanford University, December 1998.
144. N.A. Harman, “Verifying a Simple Pipelined Microprocessor Using Maude,” *15th International Workshop on Recent Trends in Algebraic Development Techniques (WADT ’01)*, M. Cerioli, and G. Reggio, *eds.*, LNCS 2267, Springer-Verlag, April 2001, pp. 128–151.
145. M.N. Lis, “Superscalar Processors via Automatic Microarchitecture Transformations,” M.S. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 2000.
146. J. Matthews, and J. Launchbury, “Elementary Microarchitecture Algebra,” *Computer-Aided Verification (CAV ’99)*, N. Halbwachs, and D. Peled, *eds.*, LNCS 1633, Springer-Verlag, July 1999, pp. 288–300.
147. J.R. Matthews, “Algebraic Specification and Verification of Processor Microarchitectures,” Ph.D. Thesis, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, October 2000.
148. L. Ivanov, “Modeling and Verification of a Pipelined CPU,” *Midwest Symposium on Circuits and Systems (MWSCAS ’02)*, August 2002.
149. S. Ramesh, and P. Bhaduri, “Validation of Pipelined Processor Designs Using Esterel Tools: A Case Study,” *Computer-Aided Verification (CAV ’99)*, N. Halbwachs, and D. Peled, *eds.*, LNCS 1633, Springer-Verlag, July 1999, pp. 84–95.
150. P. Mishra, H. Tomiyama, N. Dutt, and A. Nicolau, "Automatic Verification of In-Order Execution in Microprocessors with Fragmented Pipelines and Multicycle Functional Units," *Design Automation and Test in Europe (DATE ’02)*, March 2002, pp. 36–43.
151. 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.
152. K. Albin, “Nuts and Bolts of Core and SoC Verification,” *38th Design Automation Conference (DAC ’01)*, June 2001, pp. 249–252.
153. N.J. Dohm, C.G. Ramey, D. Brown, S. Hildebrandt, J. Huggins, M. Quinn, and S. Taylor, “Zen and the Art of Alpha Verification,” *International Conference on Computer Design (ICCD ’98)*, October 1998.
154. F. Golshan, “Test and On-Line Debug Capabilities of IEEE Std 1149.1 in UltraSPARC-III Microprocessor,” *International Test Conference (ITC ’00)*, October 2000, pp. 141–150.
155. D.D. Josephson, S. Poehlman, V. Govan, and C. Mumford, “Test methodology for the McKinley processor,” *International Test Conference (ITC ’01)*, October–November 2001, pp. 578–585.
156. M.P. Kusko, B.J. Robbins, T.J. Koprowski, and W.V. Huott, “99% AC Test Coverage Using Only LBIST on the 1 GHz IBM S/390 zSeries 900 Microprocessor,” *International Test Conference (ITC ’01)*, October–November 2001, pp. 586–592.
157. T.L. McLaurin, and F. Frederick, “The Testability Features of the MCF5407 Containing the 4th Generation ColdFire® Microprocessor Core,” *International Test Conference (ITC ’00)*, October 2000, pp. 151–159.
158. I. Pomeranz, N.R. Saxena, R. Reeve, P. Kulkarni, and Y.A. Li, “Generation of Test Cases for Hardware Design Verification of a Super-Scalar Fetch Processor,” *International Test Conference (ITC ’96)*, October 1996, pp. 904–913.
159. M. Puig-Medina, G. Ezer, and P. Konas, “Verification of Configurable Processor Cores,” *37th Design Automation Conference (DAC ’00)*, June 2000, pp. 426–431.
160. R. Raina, R. Bailey, D. Belete, V. Khosa, R.F. Molyneaux, J. Prado, and A. Razdan, “DFT Advances in Motorola's Next-Generation 74xx PowerPC™ Microprocessor,” *International Test Conference (ITC ’00)*, October 2000, pp. 131–140.

161. S.A. Taylor, M. Quinn, D. Brown, N. Dohm, S. Hildebrandt, J. Huggins, and C. Ramey, "Functional Verification of a Multiple-Issue, Out-of-Order, Superscalar Alpha Processor," *35th Design Automation Conference (DAC '98)*, June 1998, pp. 638–643.
162. G. Vandling, "Modeling and Testing the Gekko Microprocessor, an IBM PowerPC Derivative for Nintendo," *International Test Conference (ITC '01)*, October–November 2001, pp. 593–599.
163. Intel Corporation, *The Intel® XScale™ Microarchitecture Technical Summary*, 2001.
164. H. Sharangpani, and K. Arora, "Itanium Processor Microarchitecture," IEEE Micro, September–October 2000, pp. 24–43.
165. M.N. Velev, "Using Positive Equality to Automatically Prove Liveness for Pipelined, Superscalar, and VLIW Processors with Exceptions and Branch Prediction," submitted for publication.

MIROSLAV N. VELEV has B.S.&M.S. degrees in Electrical Engineering and B.S. in Economics, received from Yale University in May 1994, and expects his Ph.D. in Electrical and Computer Engineering from Carnegie Mellon in August 2003. He has served on the program committees of ICCD, SAT, MEMOCODE, and DSD. He received the Franz Tuteur Memorial Prize for the Most Outstanding Senior Project in Electrical Engineering at Yale in 1994.