Integrating Formal Verification into an Advanced Computer Architecture Course

Miroslav N. Velev, *Member, IEEE*

***Abstract*—This paper presents a sequence of three projects on design and formal verification of pipelined and superscalar processors: 1) a single-issue, five-stage DLX (an academic processor used widely for teaching pipelined execution and defined by Hennessy and Patterson in the first edition of their graduate textbook); 2) an extension of the DLX with exceptions and branch prediction; and 3) a dual-issue superscalar DLX. The projects were integrated into two editions of an advanced computer architecture course that was offered at the Georgia Institute of Technology, Atlanta, in the summer and fall 2002 and was taught to 67 students (25 of whom were undergraduates) in a way that required them to have no prior knowledge of formal methods. Preparatory homework problems included an exercise on design and formal verification of a staggered Arithmetic Logic Unit (ALU), pipelined in the style of the integer ALUs in the Intel Pentium 4. The processors were designed and formally verified with a tool flow that was used to formally verify the M•CORE processor at Motorola and detected bugs.**

***Index Terms*—Abstraction, Boolean satisfiability (SAT), computer architecture, formal verification of microprocessors, high-level microprocessor design, logic of Equality with Uninterpreted Functions and Memories (EUFM), Positive Equality, teaching of formal methods.**

I. 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 higher complexity of new microprocessors leads to more errors—Bentley [2] reports a 350% increase in the number of bugs detected in the Intel Pentium 4 [3], compared with those detected in the previous architecture, the Intel Pentium Pro. *Formal verification* is the mathematical proof of correctness of hardware or software for all execution scenarios, as opposed to just a set of test sequences. Although industry is gradually accepting the use of formal methods, previous formal techniques did not scale for realistic processors or required extensive manual work by experts—factors that made formal methods impractical to integrate in computer architecture courses.

Traditionally, the principles of pipelined and superscalar execution have been taught with the following three approaches:

1) trace-driven simulation with existing tools [4]–[9];
2) implementation of a trace-driven simulator in a programming language such as C, often by filling only missing sections in code provided by instructors [10]–[12];
3) implementation of a processor using a commercial hardware description language (HDL), such as Verilog [13] or VHDL [14] as reported in [15]–[17], or an academic HDL [11], [18], and then simulating the design with existing tools or with a hardware emulation system based on field-programmable gate arrays (FPGAs) [12], [19]–[21].

In the second and third approaches, the implementations are verified with test sequences provided by the instructors or defined by students. However, such testing is time consuming and does not guarantee correctness—with bugs remaining undetected for years even in the skeleton code provided by the instructors [10].

In spite of the role that formal verification will play when designing computer systems in the future, only a few universities offer related courses [22]. There are many reports on the integration of formal methods into software engineering curricula [23]–[26]. However, the author knows of only one computer architecture course [10] that was offered at Carnegie Mellon University, Pittsburgh, PA, in a version where the students had to model-check [27] a cache coherence protocol.

This paper advocates the integration of formal verification into existing computer architecture courses as a way to educate students with knowledge of formal methods and with deeper understanding of the principles of pipelined, speculative, and superscalar execution. Then, when working in industry, the students will be more productive and capable of delivering correct new processors under aggressive time-to-market schedules. This paper presents the experience from such an integration of formal verification into an existing computer architecture course [28], [29], 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 the following three projects on design and formal verification of:

1) a single-issue pipelined DLX processor [30];
2) a version with exceptions and branch prediction;
3) a dual-issue superscalar DLX.

The last project was motivated by commercial dual-issue superscalar processors, such as the Intel Pentium, the Alpha 21064, the IDT RISCore5000, the PowerPC 440 Core, the Motorola MC 68060, the Motorola MPC 8560, and the MIPS III used in the Emotion Engine chip of the Sony Playstation 2 [30].

The integration of formal verification into an existing computer architecture course was made possible by a tool flow [31] that was used to formally verify a new design of the pipelined M•CORE processor at Motorola and detected bugs [32]. The pedagogical power of this tool flow is a result of the following:

1) the immediate feedback given to students—it takes 0.1 s to formally verify a single-issue pipelined DLX processor

Manuscript received March 30, 2003; revised January 24, 2004.

The author is with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: mvelev@ece.cmu.edu).

Digital Object Identifier 10.1109/TE.2004.832880

[30], 0.2 s to formally verify an extension with exceptions and branch prediction, and 15 s to formally verify a dual-issue superscalar DLX, if the Boolean satisfiability (SAT) solver siege_v4 [33] is used;
2) the full detection of bugs, when a processor is formally verified;
3) the resulting objective grading—based on correctness that is proved mathematically, as opposed to being determined by passing test sequences.

Every time the design process was shifted to a higher level of abstraction, the effect was an increase in productivity. When microprocessor designers moved from the transistor level to the gate level, and later to the register-transfer level, relying on electronic design automation (EDA) tools to bridge the gap between these levels, such an effect took place. Similarly, when programmers adopted high-level programming languages, such as FORTRAN and C, relying on compilers for translation to assembly code, the same effect took place. The abstract hardware description language (AbsHDL) [31] that was used in the projects 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. These characteristics of AbsHDL allow the microprocessor designers to focus entirely on partitioning the functionality among pipeline stages and on defining the processor control logic. Most importantly, this high-level definition of processors, coupled with certain modeling restrictions (Section III), allows the efficient formal verification of the pipelined designs. The assumption is that the bit-level descriptions of functional units and memories will be formally verified separately from the rest of the circuit and will be added by EDA tools later when an AbsHDL processor is translated automatically to a bit-level synthesizable description, e.g., in Verilog or VHDL.

The rest of the paper is organized as follows. Section II summarizes related work. Section III presents the formal verification background and the tool flow. Section IV describes the three projects, and Section V discusses their integration into an existing advanced computer architecture course. Section VI presents results, and Section VII concludes the paper.

II. Related Work

Traditionally, the principles of pipelined and superscalar execution have been taught by using trace-driven simulation [4]–[12] or FPGA-based hardware emulation [12], [19]–[21]. Surveys of simulation resources are presented in [34] and [35]. However, simulator bugs and modeling inaccuracies can lead to significant errors in performance measurements [36]–[39] and therefore may result in wrong design decisions. To avoid bugs, Weaver *et al.* [8] extended their trace-driven simulator with a dynamic checker, which is similar to the checker processor in the Dynamic Implementation Verification Architecture [40] and is used to compare the superscalar simulator results with those produced by a nonpipelined simulator. Although such a checker can detect bugs triggered by the benchmarks simulated so far, it does not guarantee correctness; thus, bugs may still remain to be activated by other benchmarks. The lack of guarantee for correctness diminishes the pedagogical power of simulation when teaching the principles of pipelined, speculative, and superscalar execution.

Directions for integrating formal methods into software engineering courses are outlined by Almstrum *et al.* [41] and by Wing [42]. There are many reports from integrating formal verification into existing software engineering curricula [23]–[26]. A list of formal verification courses offered at various universities can be found at the Formal Methods Educational site [22]. However, none of those courses integrates formal methods into an existing computer architecture course. The only such course [10] that the author knows of was taught at Carnegie Mellon University in a version that included a project on model checking [27] a snoopy cache coherence protocol. In a different school, the instructor [43] used a formal verification tool to illustrate how to check properties of a cache coherence protocol when teaching computer architecture but did not assign projects.

Another course where the students designed dual-issue superscalar processors, using VHDL, is reported by Hamblen *et al.* [21]. However, dual-issue processors were attempted by only two of the nine groups, and one of them produced a design that passed the test sequences. Of the other seven groups that implemented single-issue pipelined processors, four produced designs that passed the test sequences.

Functional verification was taught by Ozguner *et al.* [1], but the students did not use formal verification tools. Van Campenhout *et al.* [17] studied student bugs made in Verilog designs of five-stage DLX models with branch prediction. When designing the DLX processors, those students did not describe the register files and the Arithmetic Logic Units (ALUs)—similar to the work presented in this paper—but used library modules. However, their processors were described at a lower level of abstraction and were of lower complexity. The models did not implement exceptions in addition to branch prediction or have dual-issue superscalar execution. Furthermore, the processors were not formally verified. Van Campenhout *et al.* report that their testing method detected 94% of the student errors.

III. Formal Verification Background

In the projects, the processors were implemented in AbsHDL [31] that has constructs for latches, memories, uninterpreted functions [44] (used to abstract functional units that produce word-level results), uninterpreted predicates (UPs) [44] (used to abstract functional units that produce bit-level results), equality comparators, multiplexors, and basic logic gates—AND, OR, and NOT. Signals are defined as one of two types: bit-level signals that are used to model completely the control of a processor and term-level signals that are used to abstract word-level values regardless of their actual number of bits.

The formal verification tool flow consists of: 1) the term-level symbolic simulator (TLSim) [31]; 2) the decision procedure equality validity checker (EVC) [31]; and 3) any efficient SAT solver. TLSim takes an implementation and a specification processor, defined in AbsHDL, and a simulation-command file indicating how to simulate the two processors and when to compare their architectural state elements. In symbolic simulation, the initial state of memories and latches is represented with new variables—Boolean variables for the initial state of bit-level signals and term variables for the initial state of term-level signals—introduced automatically by TLSim, thus allowing one to prove correctness for any initial state. TLSim propagates these variables through the processor logic, building symbolic expressions for the values of logic gates, uninterpreted functions, UPs, memories, and latches. The symbolic expressions are defined in the logic of Equality with Uninterpreted Functions and Memories (EUFM) [44], which gives mathematical representation for the AbsHDL constructs during symbolic simulation.

In order to compare the architectural state (consisting of all user-visible state elements, such as the PC, register file, data memory, exception-status flags, and Exception-PC) of a pipelined implementation with the architectural state of the specification, one must use an *abstraction function* that maps an implementation state (containing pipeline latches in addition to the architectural state) to an equivalent state of the architectural state elements by completing any partially executed instructions. Then, the symbolic expressions for the architectural state elements in the implementation can be directly compared for equality with the symbolic expressions for the architectural state elements in the specification. (The specification contains only architectural state elements.) In general, computing an abstraction function is a nontrivial task because the implementation contains partially executed symbolic instructions, each representing any of the instructions in the instruction set architecture (ISA) and possibly creating data and control dependencies for all subsequent instructions. In the first attempts to formally verify pipelined processors during the late 1980s and early 1990s, the verification engineers defined the abstraction function manually—a time-consuming task whose complexity increases with that of the pipelined implementation. A breakthrough was made with Burch and Dill's idea [44] to compute the abstraction function automatically by *flushing* the pipelined implementation—feeding it with bubbles (i.e., combinations of control signals that do not modify any architectural state element) until all partially executed instructions that are initially in the pipeline are completed, and their results are reflected on the architectural state elements. To flush a pipelined implementation, one has to incorporate a flush signal that, if asserted to *true* (i.e., 1), will turn any newly fetched instructions into bubbles and will not allow the PC to be incremented by the fetch mechanism to point to the sequential instruction but will allow instructions that are already in the pipeline to complete and update the PC and other architectural state elements. Incorporating such a flush signal in the implementation can be viewed as design for formal verification.

In the projects, symbolic simulation with TLSim was done to prove the *safety property* of a pipelined implementation processor. That one step of the implementation, starting from an arbitrary initial state, corresponds to between 0 and k steps of the specification, where k is the issue width of the implementation. To build a formula for this correctness condition, the implementation is simulated symbolically for one step from an arbitrary initial state and then flushed. The resulting symbolic expressions for the architectural state elements are compared for equality with the symbolic expressions for the same architectural state elements after first flushing the implementation and then using that architectural state to simulate symbolically the specification for 0 to k steps. If a processor is correct for one step from an arbitrary initial state, then by induction, the processor will be correct for any number of steps (see [45] for a survey of correctness criteria).

The syntax of EUFM includes terms and formulas. Terms abstract word-level values, such as data, register identifiers, memory addresses, and the entire states of memories, and are used to model the data path of a processor. Formulas represent Boolean signals and are used to model the control path of a processor and to express the correctness condition. A term can be an uninterpreted function (UF) applied on a list of argument terms; a term variable (which can be viewed as a UF with no arguments); or an *ITE* ("if-then-else") 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*, i.e., an *ITE* operator is a mathematical representation for a multiplexor. A formula can be a UP applied on a list of argument terms, a propositional variable (a 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 formulas f_1 and f_2 based on a controlling formula f, is equivalent to $f \wedge f_1 \vee \neg f \wedge f_2$. Both terms and formulas will be referred to 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*—that equal input expressions produce equal output values. Then, whether the original functional unit is an adder or a multiplier, etc., no longer matters as long as the same UF (or UP) is used to replace it in both the implementation and the specification processor. In this way, a more general problem will be proved—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* returns the data most recently written to an equal address or otherwise the initial state of the memory for that address.

The term-level symbolic simulator TLSim has commands to compare for equality the symbolic expressions for architectural state elements in the implementation and the specification, according to the safety property, and build a resulting EUFM formula. That formula is then input into the decision procedure EVC [31] that translates the formula to an equivalent Boolean formula, which has to be a tautology for the implementation to be correct. The Boolean formula can be evaluated with any SAT procedure. The SAT solver Chaff [46] was used in the projects;

however, the recently developed SAT solver siege_v4 [33] is significantly faster. The reader is referred to [47] for an overview of recent advances in SAT solvers.

The efficiency of EVC is a result of the property of *Positive Equality* [48]. After imposing some simple restrictions on the style for describing high-level microprocessors, one gets EUFM correctness formulas where most of the terms appear only in positive (not negated) equality comparisons or as arguments to UFs and UPs. This structure of the correctness formulas allows one to treat such term variables that are syntactically distinct as not equal, thus significantly simplifying the EUFM formula, reducing the solution space and achieving orders of magnitude speedup, while still performing formal verification. The speedup is at least five orders of magnitude for a dual-issue superscalar DLX processor with exceptions, multicycle functional units, and branch prediction [49].

To exploit Positive Equality, a microprocessor designer has to follow simple restrictions [49] when defining the high-level microprocessors. First, equality comparators between data operands should be abstracted with a new UP in both the implementation and the specification. Second, the data memory should be abstracted with a finite-state machine (FSM) model of a memory so that the interpreted functions *read* and *write* (satisfying the forwarding property of the memory semantics) are replaced by new uninterpreted functions f_r and f_w, respectively, that take the same arguments but do not satisfy the forwarding property. Then one would only prove that the implementation and the specification perform the same sequence of memory operations with the same argument terms, but that proof is sufficient for processors that do not reorder the memory operations, as in the case of the models that are formally verified in the projects.

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

The projects went through two iterations—one in the summer [28] and another in the fall [29] of 2002. During the summer, the students were prepared for the projects with two lectures on formal verification. Analysis of frequent student bugs and questions from the summer led to the addition of short sessions on formal verification concepts to more lectures in the fall (Section V). Preparatory homework exercises were also assigned. As a result, the students asked fewer questions when implementing the projects in the fall, although the most frequent bugs (see [50] for detailed descriptions of the bugs) were similar to those in the summer. The fall editions of the projects were slightly modified by changing the semantics of some of the instruction types or by requiring the students to distribute the functional units differently across the two execution pipelines of the superscalar processor in Project 3. These modifications made it difficult for the students to reuse designs from the summer and led to bugs specific to the fall edition of the projects.

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 students were required to complete each step before going on—the pedagogical motivation was to simplify the debugging and to ensure a clear understanding of how the various instruction types and control mechanisms interact. The correct semantics of all the functionality implemented up to and including the new step was defined with a nonpipelined specification processor that was given to the students. They were also given the TLSim simulation-command files for all steps of Project 1 but were asked to create their own simulation-command files for Projects 2 and 3. The pedagogical motivation was to reduce the number of new concepts in Project 1, where the students had to master the syntax of AbsHDL, the modeling techniques that allow efficient formal verification of high-level microprocessors, and to design correctly a pipelined processor with many control mechanisms. Furthermore, only small changes were required to convert the simulation-command file from the last step of Project 1 to simulation-command files for Projects 2 and 3, again allowing the students to focus on the correct design of new features in those projects. For a more careful understanding of the various control mechanisms in a processor, and for a more permanent effect from the projects, the students were required to describe the processors completely, as opposed to just filling in the missing parts in provided skeleton code. The following versions of the projects were assigned in the fall of 2002.

A. 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 a three-stage ALU pipeline to a five-stage pipelined processor that has 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 was to be placed in ID; and the ALU and the forwarding logic were to be in EX.

Step 2) *Implementation of register–immediate ALU instructions*. The students had to integrate a multiplexor to select an immediate data value as ALU input.

Step 3) *Implementation of store instructions*. The FSM model for abstraction of the data memory (Section III) 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 was actually used.

Step 5) *Implementation of (unconditional) jump instructions*. The students were given the hint that during flushing, the PC should be updated only by instructions that were already in the pipeline since the purpose of flushing was to complete such instructions without fetching new ones. This mechanism for updating the PC was viewed as design for formal verification.

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

B. 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 with ALU exceptions. The new specification had two additional architectural state elements—an Exception PC (EPC) that held the address of the last instruction that raised an ALU exception and a flag IsException that indicated whether an ALU exception was actually raised, i.e., whether the EPC contained valid information. A UP was used to check for ALU exceptions by having had the same inputs as the ALU and having produced a Boolean signal that indicated whether an exception was raised [51].

Step 2) *Implementation of a return-from-exception instruction*. This instruction updated the PC with the value of the EPC if flag IsException was set and then cleared that flag.

Step 3) *Implementation of branch prediction*. Since branch prediction was a mechanism that enhanced 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 an FSM [51] that produced an arbitrary term for the predicted target and an arbitrary Boolean signal for the predicted direction. 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 the branch predictor. The students were given the hint that since the purpose of flushing was to complete instructions that were already in the pipeline, the PC should not be updated speculatively by newly fetched instructions during flushing but should still be updated by the logic for correcting branch and jump mispredictions. This mechanism for PC updating was viewed as design for formal verification.

C. 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 would execute all instruction types, and the second pipeline would execute only register–register and register–immediate ALU 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.

Step 2) *Adding jump and branch instructions to the second pipeline*. The students were given the hint that the tricky part was to cancel all instructions that follow a jump or a taken branch. Furthermore, the instructions had to be canceled for all their transitions.

Project 3 was defined as an extension of Project 1 in order to introduce one difficulty at a time—superscalar execution. Thus, the combination with exceptions and branch prediction was avoided.

V. Integration of the Three Projects Into an Advanced Computer Architecture Course

The three projects were integrated into an advanced computer architecture course that was listed as both an undergraduate course (ECE 4100) and a graduate course (ECE 6100). The students were required to have taken an introductory computer architecture course; no additional prerequisites were included for a formal verification background. The course was based on the graduate textbook by Hennessy and Patterson [30]. This discussion is about the fall 2002 version of the course [29]. The slides developed at the University of California, Berkeley [52] were extended with two lectures on formal verification of processors and several short topics added to existing lectures.

To prepare the students for the lectures on formal verification, several concepts were introduced in 10–15-min 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 an example three-stage pipelined processor and its nonpipelined specification. The two lectures on formal verification of processors followed next (lectures 6 and 7 [29]) and introduced the logic of EUFM, the inductive correctness criterion for the safety property, the formal verification tool flow, the property of Positive Equality, and the modeling restrictions necessary to exploit that property. Another short session was included in the lecture on branch prediction in order to discuss the abstraction of a branch predictor with an FSM and the integration of branch prediction into 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, such 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 symbolic expressions for small circuits (consisting of two to three 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. (The last part of this exercise was motivated by student bugs from the projects in summer 2002 [28].) 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 problem was to implement the issue

TABLE I
STATISTICS FROM THE THREE PROJECTS

	Project 1	Project 2	Project 3
Number of groups	24	24	17
Min. time to complete the project per group	11 h	8.15 h	12 h
Min. time to complete the project by an undergraduate working alone	18.5 h	12 h	—
Average time to complete the project per group	22.9 h	27.5 h	29.1 h
Typical length of an implementation processor in AbsHDL lines	400–450	650–750	900–1,000
Typical formal verification time, using the SAT solver siege_v4 [33]	0.1 sec	0.2 sec	15 sec
Total number of different bugs	44	29	20

logic of a three-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), so 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 four-stage pipelined ALU, implemented in the style of the staggered integer ALUs in the Intel Pentium 4 [3]. 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, one can compute the two halves of a result in different pipeline stages, producing each half with an ALU that is half of the original width and thus can be clocked twice as fast. 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. A similar exercise can be found in the textbook by Shen and Lipasti [53], except that the students are not given a way to formally verify their implementations of a staggered ALU.

Another homework problem was assigned in summer 2002 [28]. The students were given an incorrect version of the pipelined DLX from Project 1 and were told that the processor has five bugs. The students had to inspect the code, identify the bugs, and explain each of them.

In fall 2002, Project 1 had to be completed in two weeks, while Projects 2 and 3 were assigned for three weeks each. (In summer 2002, the project durations were slightly shorter, since summer terms are 11 weeks long, compared with 15 weeks for regular terms at the Georgia Institute of Technology, Atlanta.) The students could work alone or in groups of up to three on Projects 1 and 2, and were required to work in groups of three on Project 3 because of its complexity.

VI. Results

Table I summarizes statistics from the three projects. The reported formal verification times were measured on a Dell OptiPlex GX260 having a 3.06-GHz Intel Pentium 4 processor with a 512-KB on-chip L2 cache, 2 GB of memory, and running Red Hat Linux 9. The memory required by the tool flow for these projects is less than 256 MB. (If the tool flow runs out of memory, it will exit and print an error message.) The formal verification time for Project 3 can be reduced to 0.5 s if the students are taught how to compute the abstraction function by controlled flushing [54]; that concept was not covered. The total numbers of different bugs do not include AbsHDL syntax errors; TLSim simulates only if a processor is free of syntax errors.

The reader is referred to [50] for detailed descriptions of all the bugs in the projects. Students who took less time to finish the projects consistently had the highest grades on the exams.

VII. Conclusion

The experience presented in this paper indicates 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. An efficient tool flow allowed students to design pipelined and superscalar processors in a sequence of three projects and to formally verify the designs in a few seconds. The formal verification of the above processors takes 0.1, 0.2, and 15 s, respectively. 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. Integration of formal verification into computer architecture courses will produce future microprocessor designers with a deeper understanding of the principles of pipelined, speculative, and superscalar execution—designers 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," in *Proc. 38th DAC Conf.*, Jun. 2001, pp. 253–255.

[2] B. Bentley, "Validating the Intel Pentium 4 microprocessor," in *Proc. 38th Design Automation Conf. (DAC '01)*, Jun. 2001, pp. 244–248.

[3] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the Pentium 4 processor," *Intel Technology J.*, 1st Quarter 2001.

[4] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," University of Wisconsin-Madison, Computer Sciences Tech. Rep. 1342, Jun. 1997.

[5] T. Diep, "VMW: A visualization-based microarchitecture workbench," Ph.D. dissertation, Dept. of Electrical and Computer Engineering, Carnegie Mellon Univ., Pittsburgh, PA, Jun. 1995.

[6] DLXview [Online]. Available: http://yara.ecn.purdue.edu/~teamaaa/dlxview/
[7] SPIM: A MIPS R2000/R3000 Simulator, J. Larus. [Online]. Available: http://www.cs.wisc.edu/~larus/spim.html
[8] C. T. Weaver, E. Larson, and T. Austin, "Effective support of simulation in computer architecture instruction," in *Proc. Workshop Computer Architecture Education*, May 2002, pp. 48–55.
[9] WinDLX [Online]. Available: ftp://ftp.mkp.com/pub/dlx/
[10] R. E. Bryant and T. C. Mowry. (1998, Fall) CS 740: Basic Computer Systems. Carnegie Mellon Univ., Pittsburgh, PA. [Online]. Available: http://www-2.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15740-f98/www/home.html
[11] M. Brorsson, "MipsIt—A simulation and development environment using animation for computer architecture education," in *Proc. Workshop Computer Architecture Education*, May 2002, pp. 65–72.
[12] T. Tateoka, M. Suzuki, K. Kono, Y. Maeda, and K. Abe, "An integrated laboratory for computer architecture and networking," in *Proc. Workshop Computer Architecture Education*, May 2002, pp. 110–117.
[13] D. E. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, 4th ed. Norwell, MA: Kluwer, 1998.
[14] P. J. Ashenden, *The Student's Guide to VHDL*. San Francisco, CA: Morgan Kaufmann, 1998.
[15] T. C. Huang, R. W. Melton, P. R. Bingham, C. O. Alford, and F. Ghannadian, "The teaching of VHDL in computer architecture," in *Proc. Int. Conf. Microelectronics Systems Education*, Jul. 1997, pp. 133–134.
[16] SuperScalar DLX [Online]. Available: http://www.rs.e-technik.tudarmstadt.de/TUD/res/dlxdocu/SuperscalarDLX.html
[17] D. Van Campenhout, T. Mudge, and J. P. Hayes, "Collection and analysis of microprocessor design errors," *IEEE Des. Test Comput.*, vol. 17, pp. 51–60, Oct.–Dec. 2000.
[18] E. Miller and J. Squire, "Esim: A structural design language and simulator for computer architecture education," in *Proc. Workshop Computer Architecture Education (WCAE'00)*, Jun. 2000, pp. 42–48.
[19] G. Brown and N. Vrana, "A computer architecture laboratory course using programmable logic," *IEEE Trans. Educ.*, vol. 38, pp. 118–125, May 1995.
[20] 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 Trans. Educ.*, vol. 44, pp. 109–119, May 2001.
[21] J. O. Hamblen, H. L. Owen, S. Yalamanchili, and B. Dao, "An undergraduate computer engineering rapid systems prototyping design laboratory," *IEEE Trans. Educ.*, vol. 42, Feb. 1999.
[22] Formal methods educational site [Online]. Available: http://www.cs.indiana.edu/formal-methods-education/Courses/
[23] D. Garlan, "Integrating formal methods into a professional master of software engineering program," in *Proc. 8th Z User Meeting (ZUM'94)*, Jun. 1994, pp. 71–85.
[24] J. P. Gibson, "Formal requirements engineering: learning from the students," in *Proc. Australian Software Engineering Conf.*, D. Grant, Ed., 2000, pp. 171–181.
[25] A. E. K. Sobel, "Final results of incorporating an operational formal method into a software engineering curriculum," in *Proc. 29th American Society for Engineering Education (ASEE)/IEEE Frontiers in Education Conf. (FIE'99)*, Nov. 1999, pp. 13a3-3–13a3-22.
[26] G. Tremblay, "An undergraduate course in formal methods: Description is our business," in *Proc. SIGCSE Tech. Symp. Computer Science Education*, 1998, pp. 166–170.
[27] K. L. McMillan, *Symbolic Model Checking*. Norwell, MA: Kluwer, 1993.
[28] M. N. Velev. (2002, Summer) ECE 4100, Advanced Computer Architecture. Georgia Institute of Technology, Atlanta. [Online]. Available: http://users.ece.gatech.edu/~mvelev/summer02/ece4100/
[29] ——, (2002, Fall) ECE 4100/6100, Advanced Computer Architecture. Georgia Institute of Technology, Atlanta. [Online]. Available: http://users.ece.gatech.edu/~mvelev/fall02/ece6100/
[30] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Francisco, CA: Morgan Kaufmann, 2002.
[31] 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," in *Proc. Computer-Aided Verification (CAV'01)*, Jul. 2001, pp. 235–240.
[32] S. Lahiri, C. Pixley, and K. Albin, "Experience with term level modeling and verification of the M•CORE microprocessor core," in *Proc. High Level Design, Validation and Test (HLDVT'01)*, Nov. 2001, pp. 109–114.
[33] L. Ryan. Siege SAT Solver V.4. [Online]. Available: http://www.cs.sfu.ca/~loryan/personal/
[34] C. Yehezkel, W. Yurcik, and M. Pearson, "Teaching computer architecture with a computer-aided learning environment: State-of-the-art simulators," in *Proc. Int. Conf. Simulation Multimedia in Engineering Education (ICSEE'01)*, Jan. 2001.
[35] W. Yurcik, G. Wolffe, and M. Holliday, "A survey of simulators used in computer organization/architecture courses," in *Proc. Summer Computer Simulation Conf. (SCSC'01)*, Jul. 2001, pp. 524–529.
[36] B. Black and J. P. Shen, "Calibration of microprocessor performance models," *IEEE Computer*, vol. 31, pp. 59–65, May 1998.
[37] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti, "Precise and accurate processor simulation," presented at the Workshop Computer Architecture Evaluation Using Commercial Workloads, Feb. 2002.
[38] R. Desikan, D. Burger, and S. W. Keckler, "Measuring experimental error in microprocessor simulation," in *Proc. 28th Int. Symp. Computer Architecture (ISCA)*, Jul. 2001, pp. 266–277.
[39] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich, "Flash vs. (simulated) Flash: Closing the simulation loop," in *Proc. 9th Int. Symp. Architectural Support Programming Languages Operating Systems*, Nov. 2000, pp. 49–58.
[40] T. Austin, "DIVA: A dynamic approach to microprocessor verification," *J. Instruction-Level Parallelism (JILP)*, vol. 2, Jun. 2000.
[41] V. L. Almstrum, C. N. Dean, D. Goelman, T. B. Hilburn, and J. Smith. (2000, Sept.) Support for Teaching Formal Methods: Rep. of ITiCSE 2000 Working Group on Formal Methods Education. [Online]. Available: http://www.cs.utexas.edu/users/csed/FM/work/final-v5-7.pdf
[42] J. M. Wing, "Weaving formal methods into the undergraduate computer science curriculum," in *Proc. 8th Int. Conf. Algebraic Methodology Software Technology (AMAST'00)*, May 2000, pp. 2–9.
[43] L. Ivanov, "Integrating formal verification into computer organization and architecture courses," *J. Comput. Sci. in Colleges*, vol. 17, no. 3, pp. 115–124, Feb. 2002.
[44] J. R. Burch and D. L. Dill, "Automated verification of pipelined microprocessor control," in *Proc. Computer-Aided Verification (CAV'94)*, D. L. Dill, Ed., Jun. 1994, pp. 68–80.
[45] M. D. Aagaard, B. Cook, N. A. Day, and R. B. Jones, "A framework for superscalar microprocessor correctness statements," *Software Tools for Technology Transfer (STTT)*, vol. 4, no. 3, pp. 298–312, May 2003.
[46] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proc. 38th Design Automation Conf. (DAC'01)*, Jun. 2001, pp. 530–535.
[47] H. Kautz and B. Selman, "Ten challenges redux: Recent progress in propositional reasoning and search," in *Proc. Principles Practice of Constraint Programming (CP'03)*, F. Rossi, Ed., Sep.–Oct. 2003, pp. 1–18.
[48] R. E. Bryant, S. German, and M. N. Velev, "Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic," *ACM Trans. Computational Logic (TOCL)*, vol. 2, no. 1, pp. 93–134, Jan. 2001.
[49] M. N. Velev and R. E. Bryant, "Introduction to formal verification of pipelined processors by using abstraction and positive equality," *IEEE Micro*, 2005, submitted for publication.
[50] M. N. Velev, "Collection of high-level microprocessor bugs from formal verification of pipelined and superscalar designs," in *Proc. Int. Test Conf. (ITC'03)*, Oct. 2003, pp. 138–147.
[51] M. N. Velev and R. E. Bryant, "Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction," in *Proc. 37th DAC*, Jun. 2000, pp. 112–117.
[52] D. A. Patterson. CS 252, Graduate Computer Architecture. Univ. of California at Berkeley. [Online]. Available: http://www.cs.berkeley.edu/~pattrsn/252S01/index.html
[53] J. P. Shen and M. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. New York: McGraw-Hill, Jul. 2002.
[54] J. R. Burch, "Techniques for verifying superscalar microprocessors," in *Proc. 33rd Design Automation Conf.*, Jun. 1996, pp. 552–557.

Miroslav N. Velev (S'98–M'05) received the B.S. and M.S. degrees in Electrical Engineering and the B.S. degree in Economics from Yale University, New Haven, CT, all in 1994, and the Ph.D. degree in Electrical and Computer Engineering from Carnegie Mellon University, Pittsburgh, PA, in 2004.

From 2002 to 2003, he was an Instructor in the School of Electrical and Computer Engineering at the Georgia Institute of Technology, Atlanta. He has developed a highly automatic tool flow and formal verification techniques that were adopted by Motorola and used to detect bugs in the M•CORE processor. He has also developed an Efficient Memory Model for the behavioral abstraction of memory arrays in bit-level symbolic simulation; this work was adopted by Intel, Motorola, and NEC in their internal tools, and by the Synopsys and Innologic Systems in commercial tools. He has over 45 refereed publications.