Collection of High-Level Microprocessor Bugs from Formal Verification of Pipelined and Superscalar Designs

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 collection of 93 different bugs, detected in formal verification of 65 student designs that include: 1) single-issue pipelined DLX processors; 2) extensions with exceptions and branch prediction; and 3) dual-issue superscalar implementations. The processors were described in a high-level HDL, and were formally verified with an automatic tool flow. The bugs are analyzed and classified, and can be used in research on microprocessor testing.

1. Introduction

The lack of published detailed reports on microprocessor design errors has long been noted as an obstacle to research on testing [1][2][3]. This paper presents a collection of 93 different bugs, detected in a total of 65 student processors from a sequence of three projects [4][5] on high-level modeling and formal verification of: 1) single-issue pipelined DLX [6] processors; 2) extensions with exceptions and branch prediction; and 3) dual-issue superscalar implementations. The last project was motivated by commercial dual-issue superscalar processors, such as the Intel Pentium [7], the Alpha 21064 [8], the IDT RISCore5000 [9], the PowerPC 440 Core [10], the Motorola MC 68060 [11], the Motorola MPC 8560 [11], and the MIPS III used in the Emotion Engine chip of the Sony Playstation 2 [6]. The student designs were implemented in the high-level hardware description language AbsHDL [5][12] that provides constructs for abstracting functional units and memories, while fully modeling the control logic for avoiding pipeline hazards, the issue logic, and the partitioning of functionality among pipeline stages.

Van Campenhout et al. [1][3] also analyzed student bugs, made in the design of ALUs, a simple processor, 3 implementations of a pipelined DLX with branch prediction, and one x86 model. However, the 87 student errors from the 3 DLX designs were classified into 15 broad categories that are too general to allow other researchers to replicate. In contrast, this paper presents bug analysis based on 65 designs—including base DLX processors, extensions with exceptions and branch prediction, and dual-issue superscalar models—and a total of 280 student errors that are classified as 93 different kinds, each described in detail. Furthermore, most of these errors are from more complex processors that implement exceptions in addition to branch prediction, or have dual-issue superscalar execution. Additionally, these errors were detected by formal verification, so that their list is exhaustive.

Student bugs may not be entirely representative of industrial bugs, but the tool flow and abstraction techniques, used in the projects, were applied at Motorola [13] to formally verify a model of the M•CORE processor [11], detecting two bugs in the forwarding logic and one in the issue logic; multiple variants of such bugs appear among the 93 different kinds described here. Note that in a high-level HDL, where the functional units and memories are abstracted, it is the fully implemented control logic where bugs may occur.

AbsHDL is based on the logic of Equality with Uninterpreted Functions and Memories (EUFM) [14]. In EUFM, word-level values are abstracted with *terms* (see Section 2) whose only relevant property is that of *equality* with other terms. In our earlier work [15][18], we imposed restrictions on the style for describing high-level processors in order to reduce the number of terms that appear in both positive and negated equality comparisons—and are so called *g-terms* (for general terms)—while increasing the number of terms that appear only in positive (not negated) polarity—and are so called *p-terms* (for positive terms). The property of Positive Equality [15][18] allowed us to treat syntactically different p-terms as not equal when evaluating the validity of an EUFM formula, thus achieving significant simplifications and orders of magnitude speedup. The reader is referred to [19] for a correctness proof.

The formal verification is done with an automatic tool flow, consisting of: 1) the term-level symbolic simulator TLSim [12], used to symbolically simulate the implementation and specification processors, and produce an EUFM correctness formula; 2) the decision procedure EVC [12] that exploits the property of Positive Equality and other optimizations to translate the EUFM correctness formula to an equivalent Boolean formula, which has to be a tautology in order for the implementation to be correct; and 3) an efficient SAT-checker [20][21][22].

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 Automation (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.

0-7803-8106-8/03 $17.00 Copyright 2003 IEEE

2. Background

The formal verification is done by correspondence checking—comparison of a pipelined implementation against a non-pipelined specification, using flushing [14] to compute an *abstraction function*, *Abs*, mapping an implementation state to an equivalent specification state. The correctness criterion is expressed as a formula in the logic of EUFM [14], and is a *safety property* checking that all architectural state elements are updated in synchrony by either 0, or 1, or up to k instructions after each step, where k is the issue width of the implementation—see Figure 1.

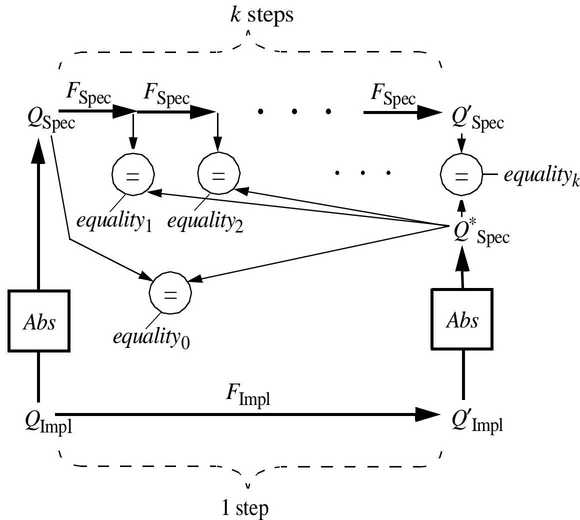

Figure 1. The safety correctness property for an implementation processor with issue width k: one step of the implementation should correspond to between 0 and k steps of the specification, when the implementation starts from an arbitrary initial state Q_{Impl}, possibly restricted by invariant constraints.

The safety property is a proof by induction, since the initial implementation state, Q_{Impl}, is 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. The reader is referred to [23][24] for a discussion of correctness criteria.

The syntax of EUFM [14] includes *terms* and *formulas*. Terms are used to abstract word-level values of data, register identifiers, memory addresses, as well as the entire states of memory arrays. A term can be an Uninterpreted Function (UF) applied on a list of argument terms, a term variable, or an *ITE* operator selecting between two argument terms based on a controlling formula, such that *ITE*(*formula*, *term1*, *term2*) will evaluate to *term1* when *formula* = *true*, and to *term2* when *formula* = *false*. The syntax for terms can be extended to model memories by means of the functions *read* and *write* [14][25]. Formulas are used to model the control path of a microprocessor, as well as to express the correctness condition. A formula can be an Uninterpreted Predicate (UP) applied on a list of argument terms, a propositional variable, an *ITE* operator selecting between two argument formulas based on a controlling formula, or an equation (equality comparison) of two terms. Formulas can be negated and connected by Boolean connectives. We will refer to both terms and formulas as *expressions*.

UFs and UPs are used to abstract the implementation details of functional units by replacing them with "black boxes" that satisfy no particular properties other than that of *functional consistency*. Namely, that the same combinations of values to the inputs of the UF (or UP) produce the same output value. 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. Thus, we will prove a more general problem—that the processor is correct for any functionally consistent implementation of its functional units. However, this more general problem is 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 efficiency from exploiting the property of Positive Equality is due to the observation that the truth (validity) of an EUFM formula under a maximally diverse interpretation of the p-terms implies the truth of the formula under any interpretation. A maximally diverse interpretation is one where the equality comparison of a term variable with itself evaluates to *true*; that of a p-term variable with a syntactically distinct term variable (a p-equation) evaluates to *false*; and that of a g-term variable with a syntactically distinct g-term variable (a g-equation) could evaluate to either *true* or *false*, and can be encoded with Boolean variables [26][27]—see [28] for a comparison of these approaches.

3. Design for Formal Verification

To enable efficient formal verification of a high-level pipelined processor, the designer must apply three restrictions:

Design for flushing. A Flush signal is introduced to disable fetching of new instructions, and to feed the pipeline with bubbles. This would allow instructions that are already in the pipeline to complete, thus using the pipeline to compute its own abstraction function [14]. During flushing, the PC should be updated with target addresses from the pipeline, e.g., due to taken branches or corrected branch mispredictions, but not from the Fetch stage, e.g., with the sequential PC formed from the current PC, or with a branch prediction.

Abstraction of equality comparisons. Equality comparisons that appear in both positive and negated polarity (e.g., as used to decide whether to take a branch-on-equal instruction, where two data operands are compared for equality, with the result appearing in positive polarity when controlling the PC

update with the branch target, but in negated polarity when squashing later instructions) are abstracted with the same uninterpreted predicate in both the implementation and the specification. This turns the data operands into p-terms, thus allowing us to more fully exploit Positive Equality.

Abstraction of the Data Memory with a conservative model. 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 2. 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 [25]. Alternatively, a hybrid memory model [25]—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—may be applied automatically. The Data Memory must be abstracted in both the implementation and the specification.

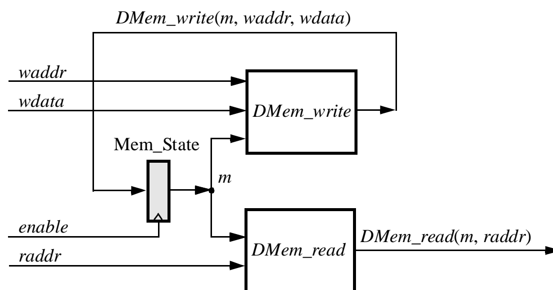

Figure 2. FSM model for conservative abstraction of memories, using uninterpreted functions *DMem_read* and *DMem_write* that do not satisfy the forwarding property of the memory semantics. Term variable *m* abstracts the initial state of the memory.

4. Description of the Three Projects

The projects went through two iterations—Summer'02 [29], and Fall'02 [4]. Each project was assigned as sequence of steps. The correct semantics of all functionality implemented up to and including the new step was defined with a non-pipelined specification, given to the students. The following versions are from Fall'02. The reader is referred to [5] for a complete description of the projects and their integration into an advanced computer architecture course.

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

Step 1.1—Implementation of register-register ALU instructions. The students were asked to extend a 3-stage pipeline 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).

Step 1.2—Implementation of register-immediate ALU instructions. The students had to integrate a multiplexor to select the immediate data value as ALU input.

Step 1.3—Implementation of store instructions. The FSM model for conservative abstraction of the Data Memory (see Figure 2) was to be added to the MEM stage.

Step 1.4—Implementation of load instructions. The students were asked to implement the load interlock in ID.

Step 1.5—Implementation of (unconditional) jumps. The jump target was to be computed in EX, and used to update the PC when the jump is in MEM.

Step 1.6—Implementation of conditional branches. The processor was to be biased for branch not taken, fetching instructions that sequentially follow a branch, and squashing them if the branch is taken.

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

Step 2.1—Implementation of ALU exceptions. The goal was to extend the processor from Step 1.6 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 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 2.2—Implementation of a return-from-exception instruction. This instruction updates the PC with the EPC value if flag IsException is set, and clears that flag.

Step 2.3—Implementation of branch prediction. Since branch prediction is a mechanism that enhances the performance of an implementation processor only, the specification was the same as for Step 2.2 of this project. The branch predictor was abstracted with an FSM [30], 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.

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

Step 3.1—Implementation of a base dual-issue DLX. The goal was to extend the processor from Step 1.4 of Project 1 to a dual-issue superscalar version where the first pipeline can execute the four instruction types of 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.

Step 3.2—Adding jump and branch instructions to the second pipeline. The specification was the same as for Step 1.6 of Project 1.

5. Bug Collection

The following are bug descriptions, frequencies, and analysis from Fall'02 [4], when the course was taken by 52 students, 13 of whom were undergraduates. A total of 280 bugs, made in the projects, were classified as 93 different kinds, of 4 categories: 1) Incorrect control logic; 2) Incorrect connections, including typos; 3) Incorrect design for formal verification; and 4) Incorrect symbolic simulation. In order to increase the readability of the AbsHDL descriptions, the students were advised to label the outputs of pipeline latches by using the name of the latch as prefix.

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 by an undergraduate working alone was 18.5 hours; the average time was 22.9 hours. A pipelined processor for Step 1.6 was implemented in 400–450 lines of AbsHDL code. Typical formal verification time for such a processor is less than 1 second.

In the specification for Step 1.6, the Instruction Memory had the outputs: SrcReg1 and SrcReg2, source register identifiers; DestReg, destination register identifier; Op, opcode; RegWrite, control bit indicating whether to update the Register File; Imm, immediate data value; use_Imm, control bit indicating whether to select the immediate data value as second operand for the ALU; MemWrite, control bit indicating whether to update the Data Memory; is_Jump, control bit indicating whether the instruction is a jump; is_Branch, control bit indicating whether the instruction is a branch; MemToReg, control bit indicating whether to select the data value loaded from the Data Memory to update the Register File. No constraints were imposed to limit the combinations of control bits, e.g., if is_Jump is *true* then is_Branch is *false*. However, no such constraints were required in the projects. Uninterpreted functions ALU and SelectTargetPC were used to abstract, respectively, the ALU and the logic that computes the target address for jumps and branches; uninterpreted predicate TakeBranchALU was used to abstract the logic that decides whether to take a branch.

The FSM for conservative abstraction of the Data Memory (see Figure 2) was implemented with uninterpreted functions DMem_Read and DMem_Update (instead of *DMem_read* and *DMem_write*, respectively, in Figure 2) that take the opcode as additional input, effectively turning them into different uninterpreted functions for every opcode—a property we call *functional non-consistency* [15] that increases the efficiency of exploiting Positive Equality. Furthermore, this allows us to model memory accesses that depend on the opcode, e.g., byte-level loads and stores.

The following is a list of distinct bugs made in each step of Project 1, sorted in descending order of their occurrences. The students made a total of 44 different bugs. AbsHDL syntax errors are not included, since TLSim simulates only if the implementation and specification are free of syntax errors, and all signals are either driven or are defined as inputs.

Step 1.1—Implementation of register-register ALU instructions

Bug 1.1.1: No forwarding path from the MEM stage to the EX stage: 7 groups

Bug 1.1.2: Incorrect forwarding logic—higher priority given to the Result in the WB stage than to the Result in the MEM stage: 5 groups

Bug 1.1.3: Read data operands from the Register File with source register identifiers from the IF stage, instead of from the ID stage: 4 groups

Bug 1.1.4: Incorrect forwarding—used the RegWrite signal from a wrong stage: 2 groups

Bug 1.1.5: Did not AND the RegWrite signal in the IF stage with signal Flush_bar to turn a newly fetched instruction into a bubble during flushing: 2 groups

Bug 1.1.6: Connected the Result from the ALU in EX to the MEM_WB latch, instead of to the EX_MEM latch: 2 groups

Bug 1.1.7: Incorrect forwarding—typo of digit 1 instead of 2 as the number of a signal in the forwarding logic: 1 group

Bug 1.1.8: Did not implement a second data operand—did not add a second output port to the Register File and a second data input to the ALU: 1 group

Bug 1.1.9: Did not implement forwarding logic for the second source operand: 1 group

Bug 1.1.10: Incorrect input to the ALU—used the opcode Op from the Instruction Memory in the IF stage, instead of from the ID_EX latch: 1 group

Bug 1.1.11: Labeled the pipeline latches with names that are different from those in the provided simulation-command file for TLSim: 1 group

Step 1.2—Implementation of register-immediate ALU instructions

Bug 1.2.1: Incorrect second data input to the MUX selecting the immediate value as ALU operand—used the value of the second operand from the ID_EX latch, instead of the output of the forwarding logic for that operand: 4 groups

Bug category	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Number of bugs in Project 1
Incorrect control logic	Bug 1.1.1 (7) Bug 1.1.2 (5) Bug 1.1.4 (2) Bug 1.1.7 (1) Bug 1.1.9 (1)	Bug 1.2.1 (4) Bug 1.2.2 (3)	————	Bug 1.4.1 (6) Bug 1.4.2 (5) Bug 1.4.3 (5) Bug 1.4.4 (4) Bug 1.4.5 (3) Bug 1.4.6 (2) Bug 1.4.9 (1) Bug 1.4.10 (1) Bug 1.4.11 (1)	Bug 1.5.2 (9) Bug 1.5.5 (4) Bug 1.5.6 (3) Bug 1.5.9 (2)	Bug 1.6.2 (4) Bug 1.6.3 (3)	76
Incorrect connections, including typos	Bug 1.1.3 (4) Bug 1.1.6 (2) Bug 1.1.8 (1) Bug 1.1.10 (1)	Bug 1.2.3 (1)	Bug 1.3.2 (6) Bug 1.3.3 (4) Bug 1.3.4 (1) Bug 1.3.5 (1)	Bug 1.4.7 (1) Bug 1.4.8 (1) Bug 1.4.12 (1)	Bug 1.5.3 (5) Bug 1.5.7 (2) Bug 1.5.8 (2) Bug 1.5.10 (1)	Bug 1.6.1 (6)	40
Incorrect design for formal verification	Bug 1.1.5 (2)	————	Bug 1.3.1 (13)	————	Bug 1.5.1 (10) Bug 1.5.4 (4)	————	29
Incorrect symbolic simulation	Bug 1.1.11 (1)	————	————	————	————	————	1
Total bugs	27	8	25	31	42	13	146

Table 1. Bug classification from Project 1. A number in parentheses indicates the occurrences of the bug.

Bug 1.2.2: Incorrect order of the inputs to the MUX selecting the immediate value: 3 groups

Bug 1.2.3: Incorrect order of the Instruction Memory outputs in the implementation: 1 group

Step 1.3—Implementation of store instructions

Bug 1.3.1: Did not AND the MemWrite signal in the IF stage with signal Flush_bar to fully turn a newly fetched instruction into a bubble during flushing: 13 groups

Bug 1.3.2: Incorrect data input to the FSM for conservative abstraction of the Data Memory—used the output from the MUX that selects the immediate value as ALU input in EX, instead of the data value that is input to that MUX: 6 groups

Bug 1.3.3: Incorrect data input to uninterpreted function DMem_Update in the FSM for conservative abstraction of the Data Memory—passed the data value from the ID_EX latch, i.e., did not account for forwarding in EX: 4 groups

Bug 1.3.4: Incorrect opcode input to the FSM for conservative abstraction of the Data Memory—used signal Op from the Instruction Memory in the IF stage, instead of from the EX_MEM latch: 1 group

Bug 1.3.5: Uninterpreted function DMem_Update, used in the FSM for conservative abstraction of the Data Memory, was mislabeled as DMem_UpDate: 1 group

Step 1.4—Implementation of load instructions

Bug 1.4.1: Did not implement path to forward the data value loaded from the Data Memory, when that value is in the WB stage: 6 groups

Bug 1.4.2: Did not fully insert a bubble in the EX stage when a load interlock is triggered—did not invalidate all control signals at the inputs of the ID_EX latch: 5 groups

Bug 1.4.3: Did not stall the PC and/or the IF_ID latch in the case of a load interlock: 5 groups

Bug 1.4.4: Incorrect load interlock condition: 4 groups

Bug 1.4.5: Did not consider stores as possible trailing instructions in the load interlock condition: 3 groups

Bug 1.4.6: Incorrect stalling condition—used signal MemToReg from IF, instead of from EX: 2 groups

Bug 1.4.7: Incorrect inputs to the AND gates for inserting a bubble in the ID_EX latch—used the signals from the outputs of the same latch: 1 group

Bug 1.4.8: Different order of the signals in the input and output ports of a pipeline latch: 1 group

Bug 1.4.9: Did not implement a load interlock: 1 group

Bug 1.4.10: Stalled the ID_EX latch instead of inserting a bubble there: 1 group

Bug 1.4.11: Inserted a bubble in the IF_ID latch, and allowed the instruction there to continue to the ID_EX latch: 1 group

Bug 1.4.12: Incorrect inputs to uninterpreted function DMem_Read in the FSM for conservative abstraction of the Data Memory: 1 group

Step 1.5—Implementation of (unconditional) jumps

Bug 1.5.1: The PC was not updated by jump instructions that are already in the pipeline during flushing: 10 groups

Bug 1.5.2: The instruction in the IF_ID latch was not squashed when that latch was simultaneously stalled by the load interlock: 9 groups

Bug 1.5.3: Incorrect order of the Instruction Memory outputs in the implementation: 5 groups

Bug 1.5.4: Did not AND the is_Jump signal in the IF stage with signal Flush_bar to fully turn a newly fetched instruction into a bubble during flushing: 4 groups

Bug 1.5.5: Did not squash all control signals for subsequent instructions, when a jump is in MEM: 4 groups

Bug 1.5.6: Did not extend the optimized load interlock to account for jumps that use a source operand: 3 groups

Bug 1.5.7: Incorrect input to uninterpreted function TargetPC_ALU that abstracts the logic for computing the target of a jump—used the opcode Op from the IF_ID latch, instead of from the ID_EX latch: 2 groups

Bug 1.5.8: Incorrect input to uninterpreted function TargetPC_ALU that abstracts the logic for computing the target of a jump—used the sequentialPC from the IF stage, instead of from the ID_EX latch: 2 groups

Bug 1.5.9: Did not AND signal EX_MEM_is_Jump with the correct phase clock to enable PC updates: 2 groups

Bug 1.5.10: No path to write the target address EX_MEM_TargetPC to the PC: 1 group

Step 1.6—Implementation of conditional branches

Bug 1.6.1: Incorrect data input to uninterpreted predicate TakeBranchALU, abstracting the logic that decides whether to take a branch—could select the immediate data value as second operand: 6 groups

Bug 1.6.2: Correctly formed the squash signal, but did not use it instead of all instances of the old squash signal, EX_MEM_is_Jump: 4 groups

Bug 1.6.3: Did not extend the load interlock signal to account for the data operand usage of branches: 3 groups

Table 1 classifies the bugs made in Project 1. From the bug descriptions and from Table 1: 59% of all mistakes in Step 1.1 were made when implementing the forwarding logic—Bug 1.1.1, Bug 1.1.2, Bug 1.1.4, Bug 1.1.7, and Bug 1.1.9. In Step 1.2, 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 (Bug 1.2.1). In Step 1.3, 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 (Bug 1.3.1). Almost as frequent were bugs due to incorrect data used as input to the Data Memory (Bug 1.3.2 and Bug 1.3.3). In Step 1.4, most frequent was Bug 1.4.1—the students did not forward 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 1.5, most frequent was Bug 1.5.1 (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 was Bug 1.5.2, where the students did not account 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 1.6, most mistakes were due to incorrect data source for the uninterpreted predicate that abstracts the branch condition (Bug 1.6.1).

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. A pipelined processor for Step 3 was implemented in 650–750 lines of AbsHDL code, depending on the amount of comments. Typical formal verification time for such a processor is less than 5 seconds.

In the specification for Step 2.2, the Instruction Memory had the same outputs as in Step 1.6 of Project 1, in addition to control bits: allow_ALU_exceptions, indicating whether an instruction can raise ALU exceptions; and is_ReturnFromException, indicating whether the instruction is a return-from-exception. Uninterpreted predicate ALU_Exception was added to abstract the logic indicating whether the ALU inputs raise an exception. Since branch prediction enhances the performance of the implementation only, the specification for Step 2.3 was the same as for Step 2.2.

The following is a list of distinct bugs made in each step of Project 2, sorted in descending order of their occurrences. Each bug is labeled with the number of the project, the number of the step, and the relative order of the bug in that step. Some groups had several bugs per step, while others had none. The students made a total of 29 different bugs when implementing Project 2. As before, AbsHDL syntax errors are not included.

Step 2.1—Implementation of ALU exceptions

Bug 2.1.1: Did not squash control bit allow_ALU_exceptions of subsequent instructions, when a jump or a taken branch is in the MEM stage: 9 groups

Bug 2.1.2: Signal EX_MEM_is_ALU_Exception was not included as part of the squash condition: 5 groups

Bug 2.1.3: Used the PC value from the IF stage as input to the EPC in the MEM stage, instead of the PC of the instruction that raised exception and is in the MEM stage: 3 groups

Bug 2.1.4: Did not allow exceptions to update the PC during flushing: 3 groups

Bug 2.1.5: Used the instruction's incremented PC as input to the EPC, instead of the instruction's PC: 2 groups

Bug 2.1.6: Did not include EX_MEM_is_ALU_ Exception as part of the enable signal for the PC: 2 groups

Bug 2.1.7: Allowed updates of the Register File and the Data Memory when the instruction raises ALU exception: 2 groups

Bug 2.1.8: Did not extend the logic for an optimized load interlock condition to account for the data operand usage of instructions that can raise exceptions: 2 groups

Bug 2.1.9: Did not extend the TLSim command file to account for the new architectural state elements: 2 groups

Bug 2.1.10: Used signals from wrong stage to enable updates of the EPC and flag IsException in the MEM stage: 1 group

Bug 2.1.11: Incorrect placement of the MUX to select the exception-handler address as input to the PC: 1 group

Step 2.2—Implementation of a return-from-exception instruction

Bug 2.2.1: EX_MEM_is_ReturnFromException was not included as part of the squash signal: 8 groups

Bug category	Step 1	Step 2	Step 3	Number of bugs in Project 2
Incorrect control logic	Bug 2.1.1 (9) Bug 2.1.2 (5) Bug 2.1.6 (2) Bug 2.1.7 (2) Bug 2.1.8 (2) Bug 2.1.11 (1)	Bug 2.2.1 (8) Bug 2.2.2 (6) Bug 2.2.3 (5) Bug 2.2.4 (4) Bug 2.2.5 (3) Bug 2.2.6 (2) Bug 2.2.7 (1) Bug 2.2.8 (1) Bug 2.2.9 (1) Bug 2.2.10 (1)	Bug 2.3.1 (14) Bug 2.3.2 (4) Bug 2.3.3 (3) Bug 2.3.4 (2) Bug 2.3.7 (1)	77
Incorrect connections, including typos	Bug 2.1.3 (3) Bug 2.1.5 (2) Bug 2.1.10 (1)	—————	Bug 2.3.6 (1) Bug 2.3.8 (1)	8
Incorrect design for formal verification	Bug 2.1.4 (3)	—————	Bug 2.3.5 (2)	5
Incorrect symbolic simulation	Bug 2.1.9 (2)	—————	—————	2
Total bugs	32	32	28	92

Table 2. Bug classification from Project 2. A number in parentheses indicates the occurrences of the bug.

Bug 2.2.2: Did not disable writes to the Register File and the Data Memory when the instruction is a return-from-exception: 6 groups

Bug 2.2.3: Return-from-exception instructions did not clear flag IsException: 5 groups

Bug 2.2.4: A return-from-exception was still completed when flag IsException is low: 4 groups

Bug 2.2.5: Incomplete logic for updating the PC with the next value of the PC: 3 groups

Bug 2.2.6: Incorrect writes to the EPC when the instruction is a return-from-exception: 2 groups

Bug 2.2.7: Incorrect placement of the MUX for updating the PC with the EPC value: 1 group

Bug 2.2.8: Did not invalidate bit is_ReturnFromException when inserting a bubble in the ID_EX latch due to the load interlock: 1 group

Bug 2.2.9: Control bit is_ReturnFromException was not squashed: 1 group

Bug 2.2.10: A return-from-exception instruction was invalidating control bits is_Jump and is_Branch for the same instruction: 1 group

Step 2.3—Implementation of branch prediction

Bug 2.3.1: Missing case in the logic for correcting branch/jump mispredictions: 14 groups

Bug 2.3.2: Did not squash the bit for the predicted-taken-direction of a branch when the branch was squashed, resulting in incorrect behavior of the logic for correcting branch mispredictions by still correcting such a branch if its prediction differs from its outcome computed in EX: 4 groups

Bug 2.3.3: Incorrect placement of the MUX for speculative updates of the PC: 3 groups

Bug 2.3.4: Allowed taken and correctly predicted branches to still squash the subsequent instructions, as was the case for taken branches in Step 2.2, where the processor had no branch prediction: 2 groups

Bug 2.3.5: Allowed speculative updates of the PC during flushing: 2 groups

Bug 2.3.6: Passed the next PC value—which could be a speculative update or a correction of a speculation—instead of the incremented PC for use in the MEM stage: 1 group

Bug 2.3.7: Incorrect speculative updates of the PC in the case of a jump in the IF stage—used the predicted direction for a branch to decide whether a jump will be taken, not considering that jumps are always taken: 1 group

Bug 2.3.8: Used signals from wrong stage as inputs to the logic for correcting a branch/jump misprediction: 1 group

Table 2 classifies the bugs made in Project 2. The most frequent bug in Step 2.1 was not squashing control bit allow_ALU_exceptions of subsequent instructions, when a jump or a taken branch is in the MEM stage (Bug 2.1.1). The second most frequent mistake was not including the ALU exception signal as part of the squash condition (Bug 2.1.2). In Step 2.2, the most frequent error was not squashing instructions that follow a return-from-exception (Bug 2.2.1). Additionally, many bugs were due to allowing architectural state updates that are disabled in the specification when control bit is_ReturnFromException is *true* (Bug 2.2.2 and Bug 2.2.3). In Step 2.3, half of the mistakes were due to missing cases in the logic for correcting branch/jump mispredictions (Bug 2.3.1). A tricky bug was not squashing the predicted direction bit of a branch, when the branch is squashed, so that the logic for correcting mispredictions would still take action to correct such a branch (Bug 2.3.2).

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. A dual-issue superscalar processor from Step 3.2 was imple-

Bug category	Step 1	Step 2	Number of bugs in Project 3
Incorrect control logic	Bug 3.1.1 (5) Bug 3.1.2 (3) Bug 3.1.3 (2) Bug 3.1.4 (2) Bug 3.1.7 (1) Bug 3.1.8 (1) Bug 3.1.10 (1) Bug 3.1.11 (1) Bug 3.1.12 (1)	Bug 3.2.1 (11) Bug 3.2.2 (3) Bug 3.2.3 (2) Bug 3.2.4 (1) Bug 3.2.5 (1) Bug 3.2.6 (1) Bug 3.2.7 (1)	37
Incorrect connections, including typos	Bug 3.1.9 (1)	Bug 3.2.8 (1)	2
Incorrect design for formal verification	————	————	0
Incorrect symbolic simulation	Bug 3.1.5 (2) Bug 3.1.6 (1)	————	3
Total bugs	21	21	42

Table 3. Bug classification from Project 3. A number in parentheses indicates the occurrences of the bug.

mented in 900–1,000 lines of AbsHDL code. Typical formal verification time for such a processor is less than 30 seconds with the SAT-checker Siege [16]—one of the winners in the SAT'03 competition [17].

The following is a list of distinct bugs made in each step of Project 3, sorted in descending order of occurrences. Each bug is labeled with the number of the project, the number of the step, and the relative order of the bug in that step. The students made a total of 20 different bugs when implementing Project 3. As before, AbsHDL syntax errors are not included.

Step 3.1—Implementation of a base dual-issue DLX

Bug 3.1.1: Incorrect priority of the results in the extended forwarding logic: 5 groups

Bug 3.1.2: The issue logic did not include a condition for a structural hazard in the ID2 stage, i.e., a check whether the instruction in ID2 is a load or a store and so requires the first pipeline: 3 groups

Bug 3.1.3: Signal issue_ID2_to_EX2 did not account for the load interlock between EX1 and ID2: 2 groups

Bug 3.1.4: Incomplete implementation of the shift register for the issue logic in ID: 2 groups

Bug 3.1.5: Did not flush the implementation processor long enough: 2 groups

Bug 3.1.6: Did not simulate the specification processor for two cycles: 1 group

Bug 3.1.7: Did not AND signal issue_ID2_to_EX2 with the RegWrite signal that goes from ID2 to EX2, in order to conditionally issue the instruction in ID2 to EX2: 1 group

Bug 3.1.8: The data value loaded from the Data Memory could not be forwarded from WB: 1 group

Bug 3.1.9: Wrong order of the Instruction Memory outputs in the implementation: 1 group

Bug 3.1.10: Did not consider a load interlock between the instructions in ID1 and ID2: 1 group

Bug 3.1.11: Did not insert bubble in EX1 and EX2 when neither of the instructions in the ID stage is issued: 1 group

Bug 3.1.12: Did not insert a bubble in EX2 when the instruction in ID2 is not issued: 1 group

Step 3.2—Adding jump and branch instructions to the second pipeline

Bug 3.2.1: Did not squash the instruction in ID2 when that instruction is shifted to ID1 and there is a jump or a taken branch in the MEM2 stage: 11 groups

Bug 3.2.2: Did not squash the instructions in the ID stage when both of them are stalled and there is a jump or a taken branch in the MEM2 stage: 3 groups

Bug 3.2.3: Did not insert bubbles in the EX1 and EX2 stages when neither of the instructions in ID is issued: 2 groups

Bug 3.2.4: Did not extend the load interlock signal with control bits is_Jump and is_Branch: 1 group

Bug 3.2.5: Did not squash the instruction in the ID1 stage when it is a branch/jump that is steered to the second pipeline, and there is a jump or a taken branch in MEM2: 1 group

Bug 3.2.6: Did not implement squashing logic: 1 group

Bug 3.2.7: Incorrect logic gate—used an AND instead of an OR gate—in the logic for signal issue_ID1: 1 group

Bug 3.2.8: Used control bit is_Branch from ID2 to check if the instruction in ID1 is a branch: 1 group

Table 3 classifies the bugs made in Project 3. The most frequent bug in Step 3.1 was incorrect priority of the results forwarded to the two ALUs in EX (Bug 3.1.1). The rest of the bugs in Step 3.1 were due to incorrect implementation of the issue logic based on a shift register. In Step 3.2, most frequent was Bug 3.2.1—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.

6. Analysis of the Bugs in the Three Projects

The students, working in groups, reported a total of 280 bugs when completing the three projects. The bugs were classified as being of 93 different kinds—see the detailed bug descriptions for each project—and of 4 categories—see Tables 1, 2,

Bug category	Project 1	Project 2	Project 3	Sequence of three projects
Incorrect control logic	52%	84%	88%	68%
Incorrect connections, including typos	27%	9%	5%	18%
Incorrect design for formal verification	20%	5%	0%	12%
Incorrect symbolic simulation	0.7%	2%	7%	2%
Total bugs	146	92	42	280

Table 4. Frequency of the bug categories in the three projects.

and 3. Table 4 summarizes the frequency of each category in the three projects.

As Table 4 shows, the relative frequency of the second and third bug categories—"Incorrect connections, including typos," and "Incorrect design for formal verification"—decreased dramatically from Project 1 to Project 3—from 27% and 20%, respectively, in Project 1 to 5% and 0%, respectively, in Project 3. That is, the total share of these two bug categories decreased from 47% in Project 1 to 5% in Project 3. This can be explained with the increased experience of the students with both AbsHDL and the concept of design for formal verification. As a result, the bugs in Projects 2 and 3 were exclusively of the first category—Incorrect control logic—which accounted for 84% and 88%, respectively. The increased share of the fourth bug category, Incorrect symbolic simulation, in Project 3—7%, compared to 0.7% in Project 1, and 2% in Project 2—is due to the new requirements to flush a dual-issue superscalar DLX longer than a single-issue implementation, and to simulate the non-pipelined specification for two steps to account for the case of the superscalar processor fetching two new instructions in a step and later completing both. Most importantly, the total number of bugs decreased from 146 in Project 1 to 42 in Project 3, as the students became more skillful in correctly designing pipelined processors. The requirement that the students work in groups of three on Project 3 (they could work alone, or in groups of two on Projects 1 and 2) probably also helped reduce the bugs. Note that the third and fourth bug categories—"Incorrect design for formal verification" and "Incorrect symbolic simulation"—did not exist previously, when processors were verified with binary sequences.

7. Related Work

Van Campenhout et al. [1][3] classified 87 student bugs—made in 3 Verilog designs of 5-stage DLX processors, each implemented by a single student, and having branch prediction—into 15 broad categories: wrong signal source, missing instance, missing inversion, new category, unconnected input(s), missing input(s), wrong gate/module type, missing term/factor, wrong constant, always statement, conflicting outputs, conceptual error, signal declaration, extra term/factor, and gate or module input. However, these categories are too general to allow other researchers to replicate the bugs, or to unambiguously classify new bugs. 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 designed for formal verification; and were not formally verified—Van Campenhout et al. report that their testing method detected 94% of the student errors.

Avizienis and He [2] classified the Intel Pentium II bugs, based on publicly available errata lists [31]. However, the bug descriptions in those lists are too general to allow replication, and give only the programmer-visible manifestation of each bug, as opposed to its actual cause.

A classification of bugs detected in the Intel Pentium 4 is presented by Bentley and Gray [32][33]. Jones [34] discusses bugs found in formal verification of circuits from previous Intel processors. Lahiri et al. [13] used TLSim and EVC at Motorola. They formally verified a model of the M•CORE processor [11], and found three bugs—two in the forwarding logic, and one in the issue logic—as well as several corner cases that were not fully implemented.

In the current paper, when abstracting the bit-level implementations of functional units and memories, it is assumed that they are formally verified separately. Pandey and Bryant [35] combined Symbolic Trajectory Evaluation (STE) [36][37][38][39][40] with symmetry reductions to formally verify large memory arrays at the transistor level. Claesen et al. [41] used theorem proving to formally verify an SRT integer divider. Aagaard and Seger [42] combined model checking and theorem proving to show the correctness of an Intel multiplier. O'Leary et al. [43] formally verified an SRT integer square root circuit. Russinoff [44] proved the correctness of the multiplication, division, and square root algorithms of the AMD K7 processor. Jones et al. [34][45] combined STE with theorem proving to formally verify an instruction decoder, and a floating-point adder/subtracter from Intel processors. Chen et al. [46] 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 [47] proved the correctness of floating-point adders and multipliers. Bryant and Chen [48] formally verified large combinational multipliers. Parthasarathy et al. [49] combined ATPG with BDD-based symbolic simulation to enable efficient formal verification of the Memory Management Unit in a high-performance Motorola processor. Hazelhurst et al. [50] combined STE with SAT/BDD-based model checking to formally verify properties of a FIFO queue, and of a bus protocol from the Intel Pentium III.

Albin [51] gives an overview of microprocessor verification at Motorola, and advocates extending the design flow with the high-level definition and formal verification of pipelined processors, before their automatic synthesis. Puig-

Medina et al. [52] present the testing methodology for configurable processor cores at Tensilica.

8. Conclusions

The paper presented a collection of 93 different bugs, detected in formal verification of 65 high-level student designs that include: 1) single-issue pipelined DLX processors; 2) extensions with exceptions and branch prediction; and 3) dual-issue superscalar versions. The bugs were analyzed, and classified into 4 categories, including two that did not exist in previous taxonomies—Incorrect design for formal verification, and Incorrect symbolic simulation. The detailed bug descriptions can be used by researchers developing microprocessor testing techniques, as well as by instructors who will adopt the sequence of three projects.

References

[1] 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.

[2] A. Avizienis, and Y. He, “Microprocessor Entomology: A Taxonomy of Design Faults in COTS Microprocessors,” *Dependable Computing for Critical Applications (DCCA ’99)*, January 1999, pp. 3–23.

[3] 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.

[4] M.N. Velev, Georgia Institute of Technology, *ECE 4100/6100, Advanced Computer Architecture*, Fall 2002, http://users.ece.gatech.edu/~mvelev/fall02/ece6100/.

[5] M.N. Velev, “Integrating Formal Verification into an Advanced Computer Architecture Course,” *ASEE Annual Conference & Exposition*, June 2003.

[6] J.L. Hennessy, and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd edition, Morgan Kaufmann Publishers, 2002.

[7] Intel Corporation, Intel Pentium® Processor, http://developer.intel.com/design/pentium.

[8] Digital Equipment Corporation, *Digital Semiconductor Alpha 21064 and 21064A Microprocessors: Hardware Reference Manual*, June 1996.

[9] IDT, Inc., *IDT 79RC5000™ RISC Microprocessor Reference Manual*, September 2000.

[10] IBM, Inc., PowerPC 440, http://www-3.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_440_Embedded_Core.

[11] Motorola, Inc., Semiconductor Products, http://e-www.motorola.com.

[12] 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, pp. 235–240.

[13] 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.

[14] 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.

[15] M.N. Velev, and R.E. Bryant, “Exploiting Positive Equality and Partial Non-Consistency in the Formal Verification of Pipelined Microprocessors,” *36th Design Automation Conference (DAC ’99)*, June 1999.

[16] L. Ryan, Siege SAT Solver v.3, http://www.cs.sfu.ca/~loryan/personal/.

[17] D. Le Berre, and L. Simon, “Results from the SAT Solver Competition,” *6th International Conference on Theory and Applications of Satisfiability Testing (SAT ’03)*, May 2003. http://www.lri.fr/~simon/contest03/results/.

[18] M.N. Velev, and R.E. Bryant, “Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic,” *Correct Hardware Design and Verification Methods (CHARME ’99)*, L. Pierre, and T. Kropf, *eds.*, LNCS 1703, Springer-Verlag, September 1999, pp. 37–53.

[19] 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), Vol. 2, No. 1 (January 2001), pp. 93–134.

[20] 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.

[21] 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.

[22] E. Goldberg, and Y. Novikov, “BerkMin: A Fast and Robust Sat-Solver,” *Design, Automation, and Test in Europe (DATE ’02)*, March 2002.

[23] 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.

[24] 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.

[25] M.N. Velev, “Automatic Abstraction of Memories in the Formal Verification of Superscalar Microprocessors,” *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2031, Springer-Verlag, 2001.

[26] 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)*, LNCS 1427, Springer-Verlag, June 1998.

[27] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel, “Deciding Equality Formulas by Small-Domain Instantiations,” *Computer-Aided Verification (CAV ’99)*, LNCS 1633, Springer-Verlag, June 1999, pp. 455–469.

[28] 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), Vol. 35, No. 2 (February 2003), pp. 73–106.

[29] M.N. Velev, Georgia Institute of Technology, *ECE 4100, Advanced Computer Architecture*, Summer 2002, http://users.ece.gatech.edu/~mvelev/summer02/ece4100/.

[30] 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.

[31] Intel Corporation, Pentium II Processor Specification Update, April 1999, http://www.intel.com/design/pentiumii/specupdt/.

[32] B. Bentley, and R. Gray, “Validating the Intel® Pentium® 4 Processor,” Intel Technology Journal, 1st Quarter, 2001.

[33] B. Bentley, “Validating the Intel® Pentium® 4 Microprocessor,” *38th Design Automation Conference (DAC ’01)*, June 2001, pp. 244–248.

[34] R.B. Jones, *Symbolic Simulation Methods for Industrial Formal Verification*, Kluwer Academic Publishers, Boston/Dordrecht/London, 2002.

[35] 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.

[36] D.L. Beatty, “A Methodology for Formal Hardware Verification with Application to Microprocessors,” Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, August 1993.

[37] Intel Corporation, *Partial Bibliography of STE Related Research*, http://intel.com/research/scl/library STE_Bibliography.pdf.

[38] A. Jain, “Formal Hardware Verification by Symbolic Trajectory Evaluation,” Ph.D. Thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, August 1997.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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,” *Theorem Provers in Circuit Design: Theory, Practice and Experience (TPCD ’94)*, LNCS 901, Springer-Verlag, 1995, pp. 52–71.

[44] 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.

[45] 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).

[46] 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, pp. 19–33.

[47] 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.

[48] 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.

[49] G. Parthasarathy, M. K. Iyer, T. Feng, L.-C. Wang, K.-T. Cheng, M. S. Abadir. Combining ATPG and Symbolic Simulation for Efficient Validation of Embedded Array Systems,” *International Test Conference (ITC ’02)*, October 2002.

[50] S. Hazelhurst, O. Weissberg, G. Kamhi, and L. Fix, “A Hybrid Verification Approach: Getting Deep into the Design,” *39th Design Automation Conference (DAC ’02)*, June 2002, pp. 111–116.

[51] K. Albin, “Nuts and Bolts of Core and SoC Verification,” *38th Design Automation Conference (DAC ’01)*, June 2001, pp. 249–252.

[52] M. Puig-Medina, G. Ezer, and P. Konas, “Verification of configurable processor Cores,” *Design Automation Conference (DAC ’00)*, June 2000.