TLSim and EVC: a term-level symbolic simulator and an efficient decision procedure for the logic of equality with uninterpreted functions and memories

Miroslav N. Velev*

Consultant, USA
E-mail: mvelev@ieee.org
*Corresponding author

Randal E. Bryant

School of Computer Science,
Carnegie Mellon University,
Pittsburgh, PA 15213, USA
E-mail: Randy.Bryant@cs.cmu.edu

Abstract: We present a tool flow for high-level design and formal verification of embedded processors. The tool flow consists of the term-level symbolic simulator TLSim, the decision procedure EVC (Equality Validity Checker) for the logic of Equality with Uninterpreted Functions and Memories (EUFM), and any SAT solver. TLSim accepts high-level models of a pipelined implementation processor and its non-pipelined specification, as well as a command file indicating how to simulate them symbolically, and produces an EUFM formula for the correctness of the implementation. EVC exploits the property of Positive Equality and other optimisations in order to translate the EUFM formula to an equivalent Boolean formula that can be solved with any SAT procedure. An earlier version of our tool flow was used to formally verify a model of the M•CORE processor at Motorola, and detected bugs.

Keywords: design automation; hardware design languages; logic; microprocessors; simulation; symbolic manipulation.

Reference to this paper should be made as follows: Velev, M.N. and Bryant, R.E. (2005) ‘TLSim and EVC: a term-level symbolic simulator and an efficient decision procedure for the logic of equality with uninterpreted functions and memories’, *Int. J. Embedded Systems*, Vol. 1, Nos. 1/2, pp.134–149.

Biographical notes: Miroslav N. Velev received BS and MS in Electrical Engineering and BS in Economics from Yale University, New Haven, CT in 1994, and PhD in Electrical and Computer Engineering from Carnegie Mellon University, Pittsburgh, PA in 2004. In 2002 and 2003, he was an Instructor at the School of Electrical and Computer Engineering at the Georgia Institute of Technology in Atlanta, Georgia. He has over 45 refereed publications. He is Member of the editorial boards of the Journal of Universal Computer Science (JUCS), and the Journal on Satisfiability, Boolean Modeling and Computation (JSAT). He has served on the technical program committees of over 70 conferences, including AAAI, ASP-DAC, CADE, CASES, CHARME, DATE, ICCD, ISCAS, ISQED, MEMOCODE, RTAS, and SAT. He was an invited speaker at the *International SoC Design Conference (ISOCC’05)*, Seoul, Korea, October 2005. He is the recipient of the Franz Tuteur Memorial Prize for the Most Outstanding Senior Project in Electrical Engineering, Yale University, May 1994, and of the 2005 EDAA Outstanding Dissertation Award for the Topic New Directions in Logic and System Design.

Randal E. Bryant is Dean of the Carnegie Mellon University School of Computer Science. He has been on the faculty at Carnegie Mellon since 1984, starting as an Assistant Professor and progressing to his current rank of University Professor. His research focuses on methods for formally verifying digital hardware, and more recently some forms of software. He has received widespread recognition for his work. He is a Fellow of the IEEE and the ACM, as well as a Member of the National Academy of Engineering. His awards include the 1997 ACM Kanellakis Theory and Practice Award, as well as the 1989 IEEE W.R.G. Baker Prize for the best paper appearing in any IEEE publication during the preceding year. He received his BS in Applied Mathematics from the University of Michigan in 1973, and his PhD from MIT in 1981.

Copyright © 2005 Inderscience Enterprises Ltd.

1 Introduction

In the near future, there will be trillions of microprocessors—hundreds per person. Many of the designs will be custom tailored and highly optimised for specific applications. Most will function autonomously, without the direct supervision of humans and many will be used in safety-critical applications (Tennenhouse, 2000). Thus, it is crucial that processors be designed without errors.

The time to market for new designs will decrease, while their complexity will increase. Extensive binary simulation is already impossible. What is unique about the microprocessor industry is that bugs may require the recall and replacement of all units sold—Intel had to replace 5 million Pentium® chips (Grove, 1999) when the floating-point division error was discovered in 1994. This is unlike most other industries, where it is possible to replace or even fix only the defective component, as opposed to the entire product, hence, the high cost of microprocessor bugs—$475 million in the case of the Intel® Pentium® in 1994 (Grove, 1999), and $2.1 billion in the case of a buggy Toshiba floppy microcontroller in 1999 (Pasztor and Landers, 1999).

Every time the design of computer systems was shifted to a higher level of abstraction, productivity increased, as also observed by Jones (2002). Albin (2001) similarly advocates the adoption of higher levels of abstraction. Keutzer et al. (2000) even argue that design at higher levels of abstraction results in implementations of higher quality. The logic of Equality with Uninterpreted Functions and Memories (EUFM) (Burch and Dill, 1994)—see Section 2—allows us to abstract functional units and memories, while completely modelling the control of a processor. In our earlier work on applying EUFM to formal verification of pipelined and superscalar processors, we imposed some simple restrictions (Velev and Bryant, 1999a; 1999b) on the modeling style for defining processors, resulting in correctness formulas where most of the terms (abstracted word-level values) appear only in positive equations (equality comparisons). Such term variables can be treated as distinct constants (Velev and Bryant, 1999a; 1999b), thus significantly simplifying the EUFM correctness formulas, pruning the solution space and resulting in orders of magnitude speedup of the formal verification; we call this property *Positive Equality*. These restrictions, together with techniques to model multicycle functional units, exceptions and branch prediction (Velev and Bryant, 2000), allowed our tool flow (see Section 3) to be used to formally verify a model of the M•CORE processor at Motorola (Lahiri et al., 2001), and detected three bugs, as well as corner cases that were not fully implemented. The tool flow was also used in two editions of an advanced computer architecture course (Velev, 2005a, 2003b), where undergraduate and graduate students without prior knowledge of formal methods designed and formally verified single-issue pipelined DLX processors (Hennessy and Patterson, 2002), as well as extensions with exceptions and branch prediction, and dual-issue superscalar implementations.

Our tool flow consists of:

- the term-level symbolic simulator, TLSim, used to symbolically simulate the high-level implementation and specification processors, defined in our high-level hardware description language AbsHDL, and produce an EUFM formula for correctness of the implementation with respect to the specification;
- the decision procedure EVC that exploits Positive Equality and other optimisations to translate the EUFM correctness formula into a satisfiability-equivalent Boolean formula; and
- any efficient SAT solver, used to prove that the Boolean correctness formula produced by EVC is a tautology, i.e., the original EUFM formula is valid.

Recent dramatic improvements in SAT solvers (Moskewicz et al., 2001; Goldberg and Novikov, 2002; Ryan, 2003)—see Le Berre and Simon (2005), and Velev and Bryant (2003) for comparative studies, and Biere and Kunz (2002), Kautz and Selman (2003), and Zhang and Malik (2002) for surveys—significantly sped up the solving of Boolean formulas generated by our tool flow. However, as found in Velev and Bryant (2003), the new efficient SAT solvers would not have scaled for solving the Boolean formulas if not for the property of Positive Equality that results in at least five orders of magnitude speedup when formally verifying dual-issue superscalar processors with realistic features. Efficient translations from propositional logic to CNF (Velev, 2004a, 2004c, 2004d), exploiting the special structure of EUFM formulas produced with the modeling restrictions, resulted in additional speedup of two orders of magnitude (see Section 6.3).

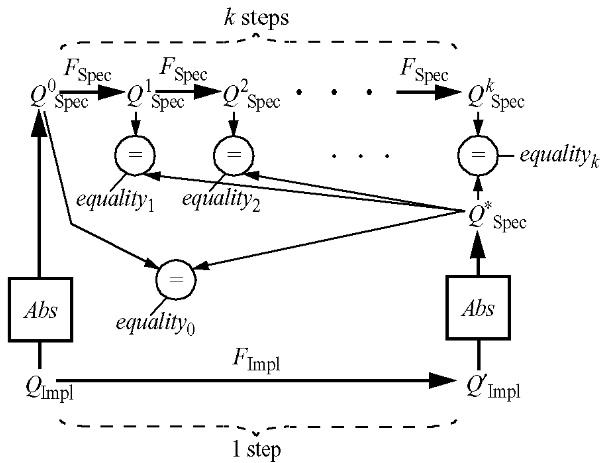
The contributions made with this paper are the high-level hardware description language AbsHDL, the term-level symbolic simulator TLSim, and the decision procedure EVC. The rest of the paper is organised as follows. Section 2 presents the background of this work. Section 3 outlines our tool flow. Section 4 defines the high-level hardware description language AbsHDL. Section 5 describes the term-level symbolic simulator TLSim. Section 6 presents the implementation of the decision procedure EVC. Section 7 summarises experimental results. Section 8 discusses related work, and Section 9 concludes the paper.

2 Background

The formal verification is done by *correspondence checking*—comparison of a pipelined implementation against a non-pipelined specification, using flushing (Burch and Dill, 1994; Burch, 1996) to automatically compute an *abstraction function* that maps an implementation state to an equivalent specification state. The safety property (see Figure 1) is expressed as a formula in the logic of EUFM, and checks whether one step of the implementation

corresponds to between 0 and k steps of the specification, where k is the issue width of the implementation. F_{Impl} is the transition function of the implementation, and F_{Spec} is the transition function of the specification. We will refer to the sequence of first applying the abstraction function and then exercising the specification as the *specification side* of the commutative diagram in Figure 1, and to the sequence of first exercising the implementation for one step and then applying the abstraction function as the *implementation side* of the commutative diagram.

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 arbitrary initial state Q_{Impl} that is possibly restricted by invariant constraints



The safety property is the inductive step of a proof by induction, since the initial implementation state, Q_{Impl}, is completely arbitrary. If the implementation is correct for all transitions that can be made for one step from an arbitrary initial state, then the implementation will be correct for one step from the next implementation state, Q'_{Impl}, since that state will be a special case of an arbitrary state, as used for the initial state, and so on for any number of steps. For some processors, e.g., where the control logic is optimised by using unreachable states as don't-care conditions, we might have to impose a set of *invariant constraints* for the initial implementation state to exclude unreachable states. Then, we need to prove that those constraints will be satisfied in the implementation state after one step, Q'_{Impl}, so that the correctness will hold by induction for that state, and so on for all subsequent states. The reader is referred to Aagaard et al. (2002; 2003) for a discussion of correctness criteria.

To illustrate the safety property in Figure 1, let the implementation and specification have three architectural state elements—program counter (PC), register file, and data memory. Let PC^i_{Spec}, $RegFile^i_{\text{Spec}}$, and $DMem^i_{\text{Spec}}$ be the state of the PC, register file, and data memory, respectively, in specification state Q^i_{Spec} ($i = 0, ..., k$) along the specification side of the diagram. Let PC^*_{Spec}, $RegFile^*_{\text{Spec}}$, and $DMem^*_{\text{Spec}}$ be the state of the PC, register file, and data memory, respectively, in specification state Q^*_{Spec}, reached after the implementation side of the diagram. Then, each disjunct $equality_i$ ($i = 0, ..., k$) is defined as

$$equality_i \leftarrow pc_i \wedge rf_i \wedge dm_i,$$

where

$$pc_i \leftarrow (PC^i_{\text{Spec}} = PC^*_{\text{Spec}}),$$

$$rf_i \leftarrow (RegFile^i_{\text{Spec}} = RegFile^*_{\text{Spec}}),$$

$$dm_i \leftarrow (DMem^i_{\text{Spec}} = DMem^*_{\text{Spec}}).$$

That is, $equality_i$ is the conjunction of the pairwise equality comparisons for all architectural state elements, thus ensuring that the architectural state elements are updated in synchrony by the same number of instructions. In processors with more architectural state elements, an equality comparison is conjuncted similarly for each additional state element. Hence, for this implementation processor, the safety property

$$equality_0 \vee equality_1 \vee \ldots \vee equality_k = true, \qquad (1)$$

is equivalently represented as

$$pc_0 \wedge rf_0 \wedge dm_0 \vee \ldots \vee pc_k \wedge rf_k \wedge dm_k = true. \qquad (1')$$

For an implementation with n architectural state elements, the safety property is

$$m_{1,0} \wedge m_{2,0} \wedge \ldots \wedge m_{n,0} \vee \ldots$$

$$\vee\, m_{1,k} \wedge m_{2,k} \wedge \ldots \wedge m_{n,k} = true, \qquad (1'')$$

where $m_{i,j}$ ($1 \le i \le n$, $0 \le j \le k$) is the condition that the state of architectural state element i after the implementation side of the diagram equals the state of that architectural state element after j steps of the specification along the specification side of the diagram.

To prove *liveness*—that the processor will complete at least one new instruction after a finite number of steps, n—we can simulate the implementation symbolically for n steps and prove that

$$equality_1 \vee equality_2 \vee \ldots\ equality_{n \times k} = true, \qquad (2)$$

omitting $equality_0$. Special abstractions and an indirect method for proving liveness, resulting in orders of magnitude speedup, are presented in Velev (2004b). Techniques for proving liveness of pipelined processors with multicycle functional units are presented in Velev (2005b).

The syntax of EUFM (Burch and Dill, 1994) includes *terms* and *formulas*—see Figure 2. Terms are used to abstract word-level values of data, register identifiers, memory addresses, as well as the entire states of memories. A term can be an Uninterpreted Function (UF) applied to a list of argument terms, a term variable, or an *ITE*

(for "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 $formula = false$. The syntax for terms can be extended to model memories by means of functions *read* and *write* (Burch and Dill, 1994; Velev, 2001). 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 to 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 combined with Boolean connectives. We will refer to both terms and formulas as *expressions*.

Figure 2 Syntax of the logic of EUFM

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

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 equal combinations of values to the inputs of the UF (or UP) produce equal output values. Then, it no longer matters whether the original functional unit is an adder, or a multiplier, etc., as long as the same UF (or UP) is used to abstract 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—but this problem is easier to prove.

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. A hybrid memory model—where the forwarding property of the memory semantics is satisfied only for those pairs of one read and one write address that also determine stalling conditions—can be applied automatically, based on rewriting rules and conservative approximations (Velev, 2001). 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. Note that the forwarding property introduces address equations in dual polarities—in positive polarity when selecting the then-expression of the *ITE*, but in negated polarity when selecting the else-expression.

The property of functional consistency of UFs and UPs can be enforced by Ackermann constraints (Ackermann, 1954), or by nested *ITE*s (Velev and Bryant, 1998c). The Ackermann scheme replaces each UF (UP) application in the EUFM formula F with a new term (Boolean) variable and then adds *external constraints for functional consistency*. For example, the UF application $g(a_1, b_1)$ will be replaced by a new term variable c_1, and another application of the same UF, $g(a_2, b_2)$, will be replaced by a new term variable c_2. Then, the resulting EUFM formula F' will be extended as $[(a_1 = a_2) \wedge (b_1 = b_2) \Rightarrow (c_1 = c_2)] \Rightarrow F'$. Note that the new formula is equivalent to $(a_1 = a_2) \wedge (b_1 = b_2) \wedge \neg (c_1 = c_2) \vee F'$, so that the new term variables, c_1 and c_2, appear in a negated equation. In the nested-*ITE* scheme, the first application of a UF will still be replaced by a new term variable c_1. However, the second will be replaced by $ITE((a_2 = a_1) \wedge (b_2 = b_1), c_1, c_2)$, where c_2 is a new term variable. A third application, $g(a_3, b_3)$, will be replaced by $ITE((a_3 = a_1) \wedge (b_3 = b_1), c_1, ITE((a_3 = a_2) \wedge (b_3 = b_2), c_2, c_3))$, where c_3 is a new term variable, and so on. UPs are eliminated similarly, but using new Boolean variables. In the general case of each scheme, the formulas that express equality of arguments of UF (UP) applications with k arguments will be conjunctions of k equations, one for each pair of corresponding arguments. To avoid creating circular dependencies when using the nested-*ITE* scheme, UFs and UPs have to be eliminated based on their topological order, i.e., all applications of a given UF (UP) have to be eliminated from the arguments of another application of the same UF (UP), before that application is eliminated. Otherwise, the equations between corresponding arguments will lead to cyclic dependency.

We can check whether an EUFM formula is valid, i.e., always *true*, either by using a specialized decision procedure such as the Stanford Validity Checker (SVC) (Burch and Dill, 1994; Jones et al., 1995; Barrett et al., 1996; Levitt and Olukotun, 1997), and the Integrated Canonizer and Solver (Filliâtre et al., 2001), or by translating an EUFM formula to a satisfiability-equivalent Boolean formula that has to be a tautology in order for the original EUFM formula to be valid. With our decision procedure, the Equality Validity Checker (EVC) (Velev and Bryant, 2001), we pursue the second approach.

The efficiency of our tool flow—consisting of TLSim, EVC, and a SAT solver—is due to the property of Positive Equality (Bryant et al., 2001) that EVC uses when translating EUFM formulas to equivalent Boolean formulas. To exploit Positive Equality, a microprocessor designer has to follow some simple restrictions (Velev and Bryant, 1999a; 1999b) when defining the high-level microprocessors. First, equality comparators between data operands—e.g., used to check whether to take a branch-on-equal instruction, such that the resulting signal is used in positive polarity when updating the PC with the

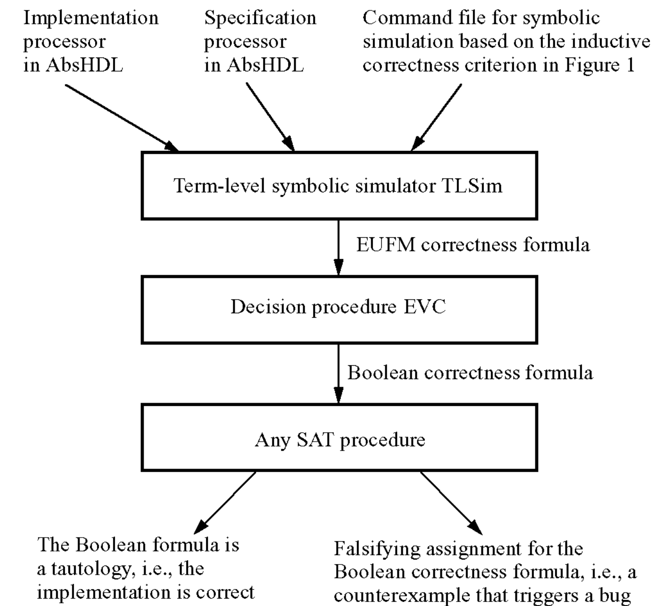
branch target address, but in negated polarity when squashing subsequent instructions—should be abstracted with a new uninterpreted predicate in both the implementation and the specification. Second, the Data Memory should be abstracted with a conservative Finite State Machine (FSM) model of a memory, where the interpreted functions *read* and *write* that satisfy 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 we would only check whether the implementation and the specification perform the same sequence of memory operations with the same argument terms, but that is sufficient for processors that do not reorder the memory operations, as is the case in the models that are formally verified in this paper.

As a result of the above restrictions, we get EUFM correctness formulas where most of the terms appear only as arguments of positive (not negated) equations, called *p-equations*, or as arguments to UFs and UPs; we call such terms p-terms. Only a few of the terms appear as arguments of equations that are used in both positive and negated polarity, and so are called g-equations (for general equations); we call such terms *g-terms*. Furthermore, when using the nested-*ITE* scheme to eliminate UF applications that appear as p-terms, we can treat the introduced new term variables as p-terms (Bryant et al., 2001). The resulting structure of the EUFM correctness formulas allows us to consider syntactically distinct *p*-term variables as not equal when evaluating the validity of an EUFM formula, thus significantly simplifying the formula, pruning the solution space, and achieving orders of magnitude speedup. The speedup is at least five orders of magnitude when formally verifying dual-issue superscalar DLX processors with realistic features—see Section 7. However, each g-equation can be either *true* or *false*, and is encoded with Boolean variables (Goel et al., 1998; Pnueli et al., 2002; Velev, 2003a) by accounting for the property of transitivity of equality (Bryant and Velev, 2002) when translating an EUFM formula to an equivalent Boolean formula.

3 The tool flow

Figure 3 summarises our tool flow. The term-level symbolic simulator TLSim (see Section 5) accepts an implementation processor and its specification, both defined in AbsHDL (see Section 4), as well as a command file indicating how to simulate the two processors symbolically according to the inductive correctness criterion in Figure 1, and outputs an EUFM correctness formula in the format of the SVC (http://sprout.Stanford.EDU/SVC). Our decision procedure EVC (see Section 6) takes an EUFM correctness formula and translates it to an equivalent Boolean formula that has to be a tautology in order for the original EUFM formula to be valid, i.e., for the implementation processor to be correct. A falsifying assignment for the Boolean correctness formula indicates a condition that triggers a bug in the implementation processor, and can be analysed to correct that bug.

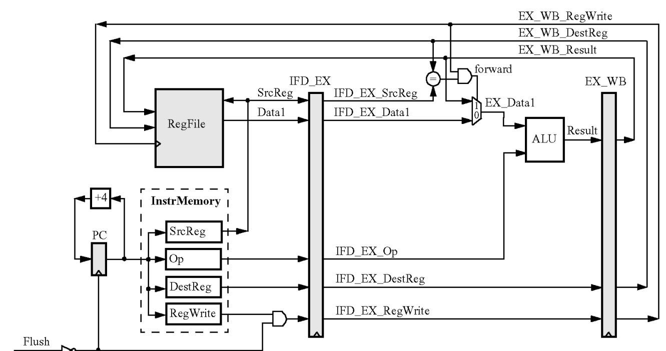
Figure 3 Our tool flow



4 The hardware description language AbsHDL

The syntax of AbsHDL will be illustrated with the three-stage pipelined processor, pipe3, shown in Figure 4. That processor has a combined instruction fetch and decode stage (IFD), an execute stage (EX), and a write-back stage (WB). It can execute only ALU instructions with a single data operand. Read-after-write hazards (Hennessy and Patterson, 2002) are avoided with one level of forwarding. The AbsHDL definition of pipe3 is shown in Figure 5. We will use the extension `.abs` for files in AbsHDL.

Figure 4 Block diagram of the three-stage pipelined processor pipe3



An AbsHDL processor description begins with declaration of signals (see Figure 5). Bit-level signals are declared with the keyword `bit`, and word-level signals with the keyword `term`. Signals that are primary inputs, e.g., phase clocks, are additionally declared with the keyword `input`. The language has constructs for basic logic gates—`and`, `or`, `not`, `mux`—such that `and` and `or` gates can have multiple inputs. Equality comparators are gates of

type =, e.g., `RegsEqual = (= IFD_EX_SrcReg EX_WB_DestReg)` in the EX stage, where the = before the left parenthesis designates an assignment, and the one after that the type of the gate. Gates that are not of the above types are uninterpreted functions if the output is a word-level signal—e.g., `sequentialPC = (PCAdder PC)` and `Result = (ALU IFD_EX_Op EX_Data1)` in Figure 5—but uninterpreted predicates if the output is a bit-level signal. Uninterpreted functions and uninterpreted predicates are used to abstract the implementations of combinational functional units. In the two examples above, `PCAdder` and `ALU` are uninterpreted functions that abstract, respectively, the adder for incrementing the PC and the ALU in pipe3. We can use an uninterpreted predicate to abstract a functional unit that decides whether to take a conditional branch, or to abstract a functional unit that indicates whether an ALU exception is raised. We can implement a Finite State Machine to model the behaviour of a multicycle functional unit (Velev and Bryant, 2000).

Figure 5 AbsHDL description of the three-stage pipelined processor pipe3

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

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

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

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

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

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

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

AbsHDL has constructs for latches and memories, defined with the keywords `latch` and `memory`, respectively. Both can have input and/or output ports, defined with the keywords `inport` and `outport`, respectively. Input ports of latches have an enable signal, which has to be high for a write operation to take place at that port, and a list (enclosed in parentheses) of input data signals that provide the values to be written to the latch. Similarly, output ports of latches have an enable signal, which has to be high for a read operation to take place at that port and a list of output data signals that will get the values stored in the latch. An output data signal can get values only from input data signals that appear in the same position in the list of data signals for an input port in the same latch. Memories are defined similarly, except that ports additionally have an address input that is listed right after the enable input—see memory `RegFile` in Figure 5.

The correct instruction semantics are defined by the Instruction Set Architecture (ISA), and are modeled with a non-pipelined specification processor built from the same uninterpreted functions, uninterpreted predicates and architectural state elements (the PC and the Register File in pipe3) as the pipelined implementation. Since the specification is non-pipelined, it lacks pipeline latches (IFD_EX and EX_WB in pipe3) and mechanisms to avoid hazards (the forwarding logic in pipe3), and executes one instruction at a time.

When defining pipelined processors and their specifications, we assume that they do not execute self-modifying code, which allows us to model the Instruction Memory as a read-only memory, separate from a Data Memory in the case of processors with load and store instructions. In Figure 5, the Instruction Memory has one read port that takes the PC as address and produces the four fields of an instruction in the given ISA: `RegWrite`, a bit indicating whether the instruction will update the Register File; `DestReg`, destination register identifier; `Op`, opcode to be used by the ALU; and `SrcReg`, source register identifier. Alternatively, a read-only instruction memory can be modelled with a collection of uninterpreted functions and uninterpreted predicates, each taking as input the instruction address and mapping it to a field from the instruction encoding. In the case when some of the above fields do not have a counterpart in the instruction encoding, but are produced by decoding logic, both models can be viewed as encapsulating the original read-only instruction memory and the decoding logic. To model decoding logic that is separate from the instruction memory, we can use uninterpreted functions and uninterpreted predicates, each mapping a field from the original instruction encoding to a control signal.

AbsHDL does not model delays of logic gates, memories and latches. It is assumed that the clock cycle will be long enough to satisfy all timing requirements.

Signal `Flush` in Figures 4 and 5, when asserted to 1, is used to disable fetching of instructions and to feed the pipeline with bubbles, allowing partially executed instructions to complete. Then, simulating the pipeline for a sufficient number of clock cycles—as determined by the pipeline depth and possible stalling conditions—will map all partially executed instructions to the architectural state elements (the PC and the Register File in pipe3). The contents of the architectural state elements, with no pending updates in the pipeline, can be directly compared with the contents of the architectural state elements of the specification. In the case of pipe3, which has two pipeline latches and no stalling logic, setting signal `Flush` to 1 and simulating the processor for two cycles will complete any instructions that are originally in the pipeline. This

mechanism for automatically mapping the state of an implementation processor to its architectural state elements was proposed by Burch and Dill (1994), and Burch (1996). Note that most processors have a similar signal indicating whether the Instruction Cache provided a valid instruction in the current clock cycle, so that we can achieve the above effect by forcing that signal to the value indicating an invalid instruction. Adding signal `Flush`—to allow completion of partially executed instructions in a pipelined or superscalar processor without fetching new instructions—can be viewed as *design for formal verification*. Signal `Flush`, when set to 1, should invalidate all control bits that indicate updates of architectural state elements.

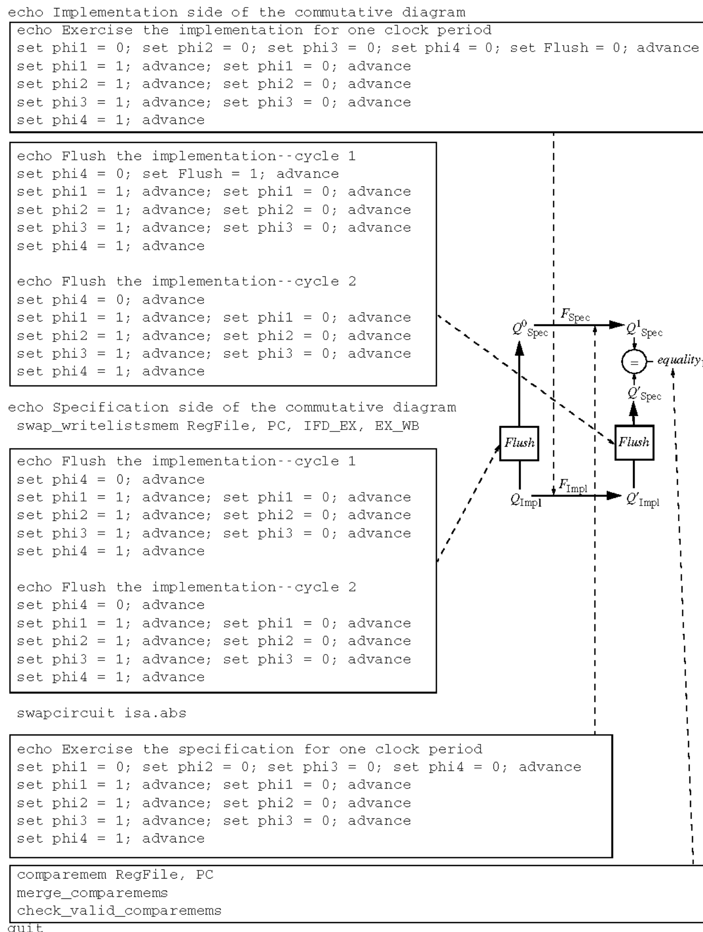
The phase clocks in an AbsHDL processor description are used to ensure the proper flow of signals in the pipeline stages, as well as to determine the order of memory port operations. In Figure 5, we assume that the phase clocks become high in the order of their numbers. Thus, in Figure 5, the pipeline latches and the PC are read on `phi1`, then the Register File is written on `phi2` and read on `phi3` (so that the Register File behaves as a write-before-read memory and provides internal forwarding of the result written in the current clock cycle), and finally the pipeline latches and the PC are written on `phi4`, which concludes a clock cycle.

We defined our own HDL, instead of using an existing commercial HDL, in order to have complete freedom in experimenting with modelling at a high level of abstraction, without having to worry about compatibility with a commercial HDL. A description in AbsHDL can be translated easily into any existing bit-level HDL. This is left for our future work.

5 The term-level symbolic simulator TLSim

To account for an arbitrary initial implementation state and for all possible transitions from it, the tool flow employs term-level symbolic simulation. (The reader is referred to Blank et al. (2001) for a discussion of symbolic simulation methods.) The term-level symbolic simulator TLSim accepts an implementation processor and its specification, both described in AbsHDL, as well as a command file indicating how to simulate the two processors symbolically and when to compare their architectural state elements, and produces an EUFM formula for correctness of the implementation with respect to the given specification. The command file for term-level symbolic simulation of pipe3 and its specification `isa.abs` with TLSim, according to the inductive correctness criterion in Figure 1, is shown in Figure 6. As discussed in Section 4, it takes two cycles to flush pipe3 after setting signal `Flush` to 1.

Figure 6 Command file for symbolic simulation of pipe3 and its specification with TLSim



```
echo Implementation side of the commutative diagram
echo Exercise the implementation for one clock period
set phi1 = 0; set phi2 = 0; set phi3 = 0; set phi4 = 0; set Flush = 0; advance
set phi1 = 1; advance; set phi1 = 0; advance
set phi2 = 1; advance; set phi2 = 0; advance
set phi3 = 1; advance; set phi3 = 0; advance
set phi4 = 1; advance

echo Flush the implementation--cycle 1
set phi4 = 0; set Flush = 1; advance
set phi1 = 1; advance; set phi1 = 0; advance
set phi2 = 1; advance; set phi2 = 0; advance
set phi3 = 1; advance; set phi3 = 0; advance
set phi4 = 1; advance

echo Flush the implementation--cycle 2
set phi4 = 0; advance
set phi1 = 1; advance; set phi1 = 0; advance
set phi2 = 1; advance; set phi2 = 0; advance
set phi3 = 1; advance; set phi3 = 0; advance
set phi4 = 1; advance

echo Specification side of the commutative diagram
swap_writelistsmem RegFile, PC, IFD_EX, EX_WB

echo Flush the implementation--cycle 1
set phi4 = 0; advance
set phi1 = 1; advance; set phi1 = 0; advance
set phi2 = 1; advance; set phi2 = 0; advance
set phi3 = 1; advance; set phi3 = 0; advance
set phi4 = 1; advance

echo Flush the implementation--cycle 2
set phi4 = 0; advance
set phi1 = 1; advance; set phi1 = 0; advance
set phi2 = 1; advance; set phi2 = 0; advance
set phi3 = 1; advance; set phi3 = 0; advance
set phi4 = 1; advance

swapcircuit isa.abs

echo Exercise the specification for one clock period
set phi1 = 0; set phi2 = 0; set phi3 = 0; set phi4 = 0; advance
set phi1 = 1; advance; set phi1 = 0; advance
set phi2 = 1; advance; set phi2 = 0; advance
set phi3 = 1; advance; set phi3 = 0; advance
set phi4 = 1; advance

comparemem RegFile, PC
merge_comparemems
check_valid_comparemems
quit
```

The term-level symbolic simulator TLSim automatically introduces new symbolic variables for the initial state of term-level or bit-level signals produced by memories and latches. TLSim propagates those variables through the processor logic, building symbolic expressions for the values of logic gates, uninterpreted functions, uninterpreted predicates, memories and latches. The symbolic expressions are defined in the script language of the SVC (http://sprout.Stanford.EDU/SVC), so that the EUFM correctness formulas produced by TLSim can be checked for validity with any EUFM decision procedure that accepts the SVC input format.

Some processors may require that their initial implementation state, Q_{Impl}, in the commutative diagram in Figure 1 be restricted by invariant constraints, to exclude unreachable states that lead to false negatives. Then, the verification engineer should check whether the invariant constraints are satisfied in the next implementation state, Q'_{Impl}. In our tool flow, the condition for each invariant constraint has to be defined as the output of an extra circuit added to the AbsHDL description of the implementation processor. (That extra circuit will be removed when an automatic tool translates the implementation into a synthesisable bit-level description.) The TLSim command `constraint`, followed by the name of a signal, allows us to use the value of that signal at the particular time step when the command appears in the simulation sequence as a constraint for checking the validity of the EUFM correctness formula. To check the invariance of a constraint in the next implementation state, the verification engineer needs to use the command `check_valid`, followed by the name of the constraint signal.

6 The decision procedure EVC

6.1 *Steps for translation from EUFM to propositional logic*

We proceed through a series of transformations, starting from an EUFM correctness formula and ending with a Boolean correctness formula that has to be a tautology in order for the original EUFM formula to be valid. At each step we apply various optimisations and simplifications. The major steps are as follows:

1. Replace equations of the form $m_1 = m_2$, where m_1 and m_2 are terms for two states of a memory, with the equation $read(m_1, a) = read(m_2, a)$, where a is a new term variable that is unique for that memory. As defined earlier, such equations can appear only as p-equations in an EUFM correctness formula checking if the two sides of the commutative diagram have updated the initial state of a memory in exactly the same way. Since the new term variable a represents an arbitrary address, if the two sides of the commutative diagram have modified that address identically, then they have modified all addresses identically. The same new term variable has to be used when replacing all equations between states of a given memory.
2. Eliminate all *read*s from updated memory states by accounting for the forwarding property of the memory semantics (see Section 2). In EVC, this step is performed dynamically, while parsing the expressions of an EUFM correctness formula. The result is that a *read* is replaced by a nested-*ITE* expression, having as a leaf a *read* from the initial state of that memory.
3. For every memory, replace each *read* from the initial state of the memory (the initial state is abstracted by a term variable that is unique for each memory) with an application of a new UF that is unique for this memory and maps an address term (argument of a replaced *read*) to a term for the initial state of that address in the given memory.
4. Classify the equations as p-equations and g-equations. Classify the terms as p-terms and *g*-terms.
5. Eliminate all UFs by using the nested-*ITE* scheme (see Section 2); classify as p-terms all new term variables introduced when eliminating a UF that was classified as a *p*-term. Eliminate all UPs by using either the nested-*ITE* or the Ackermann scheme.
6. Replace each equation that has the same term variable as both arguments with the constant *true*. Replace each p-equation between different term variables with the constant *false*, by the property of Positive Equality. Encode each g-equation with Boolean variables, by using one of the methods (Goel et al., 1998; Pnueli et al., 2002; Velev, 2003a).
7. Check if the resulting Boolean correctness formula is a tautology (or the CNF of the negated Boolean formula is unsatisfiable), which implies that the original EUFM formula is valid. Otherwise, a falsifying assignment for the Boolean correctness formula (a satisfying assignment for the CNF of the negated Boolean formula) is a condition that triggers a bug in the implementation processor.

6.2 *Hashing of expressions*

EVC uses a hashing scheme to ensure that there will be no duplicate gates and thus to increase the efficiency of SAT-checking the Boolean correctness formula. During all stages of translation from EUFM to propositional logic, the correctness formula is represented as a shared Directed Acyclic Graph, where each node is assigned an index and is identified by a unique key. The key is formed as the type of the node (*AND*, *OR*, *NOT*, *ITE*, *equation*, *read*, *write*, *uninterpreted-function*, *uninterpreted-predicate*), followed by the list of indices of the input nodes. For nodes of type *ITE*, *read*, *write*, *uninterpreted-function*, and *uninterpreted-predicate*, the order of listing the input indices is the same as the order of their nodes in the input list for the node that is being hashed. For nodes of type *equation*, the input indices are sorted in ascending order; if the two input indices are the same, then the equation node is replaced with the constant *true*. For nodes of type *AND* and

OR, the input indices are also sorted in ascending order, and duplicate input indices are removed. Furthermore, an *AND* (*OR*) node having inputs that are complements of each other (one is the negation of the other) is replaced with *false* (*true*). An *AND* node that has another *AND* node as input is replaced with a single *AND* node that has all the inputs of the two nodes, except for the eliminated node; similarly for an *OR* node that has another *OR* node as input. That is, the final Boolean formula has neither *AND* gates that directly drive other *AND* gates nor *OR* gates that directly drive other *OR* gates. Table 1 presents optimizations used when hashing expressions.

Table 1 Optimizations used when hashing expressions

Expression that is being hashed	*Retuned pointer to expression*
NOT(NOT(c))	*c*
ITE(NOT(c), a, b)	*ITE(c, b, a)*
ITE(c, a, a)	*a*
ITE(true, a, b)	*a*
ITE(false, a, b)	*b*
ITE(c, ITE(c, a, b), d)	*ITE(c, a, d)*
ITE(c, a, ITE(c, b, d))	*ITE(c, a, d)*
ITE(a, true, c)	*OR(a, c)*
ITE(a, false, c)	*AND(NOT(a), c)*
ITE(a, b, true)	*OR(NOT(a), b)*
ITE(a, b, false)	*AND(a, b)*
ITE(a, b, a)	*AND(a, b)*
ITE(a, a, c)	*OR(a, c)*

In Table 1, a chain of two *NOT*s is replaced with the input to the chain. An expression *ITE*(*NOT*(*c*), *a*, *b*), where the controlling formula is the negation of another formula *c*, is replaced with the equivalent expression *ITE*(*c*, *b*, *a*), controlled by the negation of the original controlling formula, i.e., by *c* and having the original then-input and else-input swapped. An expression *ITE*(*c*, *a*, *a*), where the then-input and the else-input are the same expression *a*, is replaced with expression *a*, since it will be selected always. If the controlling formula of an *ITE* is the constant *true* (*false*), then the *ITE* is replaced with its then-input (else-input). A chain of two *ITE*s that have the same controlling formula is replaced with one *ITE* after accounting for the truth value that has to be assigned to the controlling formula of the upper *ITE* to select the lower *ITE*, and then simplifying the lower *ITE*. That is, in *ITE*(*c*, *ITE*(*c*, *a*, *b*), *d*), the lower *ITE* will be selected when the upper *ITE*'s controlling formula *c* is *true*, so that the lower *ITE* will be equivalent to *ITE*(*true*, *a*, *b*) if selected and thus can be simplified to *a*; hence, the original chain of two *ITE*s can be replaced with *ITE*(*c*, *a*, *d*). Similarly, *ITE*(*c*, *a*, *ITE*(*c*, *b*, *d*)) can be replaced with *ITE*(*c*, *a*, *d*), since formula *c* will be *false* when the lower *ITE* is selected. The rest of the optimisations are based on the definition of an *ITE*(*c*, *a*, *b*) as $c \wedge a \vee \neg c \wedge b$ and simplifications.

In EVC, the final Boolean correctness formula consists of *AND*, *OR*, *NOT*, and *ITE* gates. EVC can evaluate that formula by using Binary Decision Diagrams (Bryant, 1986, 1992; Bryant and Meinel, 2001) via a built-in interface to the CUDD package (Somenzi, 1999, 2001), or by using Boolean Expression Diagrams (Hulgaard et al., 1999; Williams, 2000)—a non-canonical representation of Boolean functions that can be converted to Binary Decision Diagrams. Alternatively, the Boolean correctness formula can be saved to a file and then checked for being a tautology with a SAT solver. The supported formats are CNF (Johnson and Trick, 1993), ISCAS (Brglez and Fujiwara, 1985), ISCAS-CGRASP (Marques-Silva and e Silva, 1999), and Prover—a SAT solver based on Stålmarck's method (Stålmarck, 1989; Sheeran and Stålmarck, 2000). However, our comparison of SAT procedures (Velev and Bryant, 2003) determined that the most efficient way to evaluate the Boolean correctness formulas produced by EVC is with an efficient SAT solver. Chaff (Moskewicz et al., 2001) was the break-through SAT solver. Later it was surpassed by BerkMin (Goldberg and Novikov, 2002), siege (Ryan, 2003), and other tools—see Le Berre and Simon (2005) for the results from the most recent SAT solver competition.

6.3 *Efficient translation to CNF*

After the g-equations are encoded with Boolean variables, we have a purely Boolean formula that has to be a tautology for the original EUFM formula to be valid. We can check whether a Boolean formula is a tautology by negating it and proving that the resulting formula is unsatisfiable (i.e., there is no assignment to the Boolean variables that makes the formula *true*) by using any Boolean Satisfiability (SAT) solver. The most common input format of SAT solvers is Conjunctive Normal Form (CNF) (Johnson and Trick, 1993), where a Boolean formula is represented as a conjunction of *clauses*, and every clause is a disjunction of *literals*—CNF variables or their negations. In conventional translation of Boolean formulas to CNF (Tseitin, 1968), a new CNF variable is introduced for the output of every logic connective, and a set of CNF clauses is used to correlate that variable with the variables for the inputs of the connective, given the function of the connective—see Table 2. Then, the CNF for a Boolean formula is the conjunction of the clauses for all gates, conjuncted with the 1-literal clause expressing the condition that the CNF variable for the output of the formula should be *true*. Tseitin (1968) was among the first to use such CNF translation, which was later optimised by Plaisted and Greenbaum (1985), and used for testing of digital circuits by Larrabee (1992).

Table 2 Conventional translation of basic logic gates to CNF

Logic gate	*Equivalent constraints*	*CNF clauses*
$o \leftarrow \text{AND}(i_1, i_2, ..., i_n)$	$\neg i_1 \Rightarrow \neg o$	$(i_1 \vee \neg o) \wedge$
	$\neg i_2 \Rightarrow \neg o$	$(i_2 \vee \neg o) \wedge$
	...	...
	$\neg i_n \Rightarrow \neg o$	$(i_n \vee \neg o) \wedge$
	$i_1 \wedge i_2 \wedge ... \wedge i_n \Rightarrow o$	$(\neg i_1 \vee \neg i_2 \vee ... \vee \neg i_n \vee o)$
$o \leftarrow OR(i_1, i_2, ..., i_n)$	$i_1 \Rightarrow o$	$(\neg i_1 \vee o) \wedge$
	$i_2 \Rightarrow o$	$(\neg i_2 \vee o) \wedge$
	...	...
	$i_n \Rightarrow o$	$(\neg i_n \vee o) \wedge$
	$\neg i_1 \wedge \neg i_2 \wedge ... \wedge \neg i_n \Rightarrow \neg o$	$(i_1 \vee i_2 \vee ... \vee i_n \vee \neg o)$
$o \leftarrow ITE(i, t, e)$	$i \wedge t \Rightarrow o$	$(\neg i \vee \neg t \vee o) \wedge$
	$i \wedge \neg t \Rightarrow \neg o$	$(\neg i \vee t \vee \neg o) \wedge$
	$\neg i \wedge e \Rightarrow o$	$(i \vee \neg e \vee o) \wedge$
	$\neg i \wedge \neg e \Rightarrow \neg o$	$(i \vee e \vee \neg o)$
$o \leftarrow NOT(i)$	$i \Rightarrow \neg o$	$(\neg i \vee \neg o) \wedge$
	$\neg i \Rightarrow o$	$(i \vee o)$

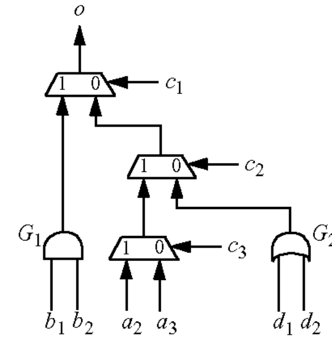
CNF-based SAT solvers face two main hurdles to further improvements. First, the operation-intensive *Boolean Constraint Propagation*—reflecting a CNF variable's assignment on all the clauses containing that variable or its negation—takes up to 90% of the SAT-solving time (Moskewicz et al., 2001) and generates many non-sequential memory accesses that are prone to L2-cache misses. Furthermore, Boolean Constraint Propagation requires data-dependent branches that are hard to predict and so frequently incur the branch misprediction penalty—at least 19 cycles, and up to 125 instructions in the Intel Pentium 4 (Hennessy and Patterson, 2002). Second, many L2-cache misses occur for big formulas (Zhang and Malik, 2003), resulting in expensive accesses to main memory; the L2-cache miss penalty is up to hundreds of cycles currently and is increasing (Hennessy and Patterson, 2002).

To reduce the above two hurdles when translating to CNF, we can preserve the *ITE*-tree structure of equation arguments, instead of replacing each equation with a disjunction of conjunctions of formulas, as done in Bryant et al. (2001), Velev and Bryant (1999b), Velev and Bryant (2001). For example, the equation $ITE(c_1, a_1, a_2) = ITE(c_2, b_1, b_2)$ will be replaced with $ITE(c_1, ITE(c_2, a_1 = b_1, a_1 = b_2), ITE(c_2, a_2 = b_1, a_2 = b_2))$, as done in Velev (2004d). This results in Boolean correctness formulas with *ITE*-trees, where each *ITE* inside a tree has *fanout count* of 1, i.e., drives only one gate that is another *ITE* inside the same tree.

An *ITE*-tree can be translated into CNF with a unified set of clauses (Velev, 2004d), without intermediate variables for outputs of *ITE*s inside the tree—see the paths from inputs a_2 and a_3 to the output o in Figure 7. Furthermore, *ITE*-trees can be merged with one level of their AND/OR leaves, where each leaf has a fanout count 1—see the paths from G_1 and G_2 to o in Figure 7. We can similarly merge *ITE*-trees with two levels of their leaves. And we can also merge other gate groups (Velev, 2004c), e.g., *AND-ITE*, *OR-ITE*, *AND-OR*, *ITE-OR*, *OR-AND*, and *ITE-AND*, but this results in only small additional improvements if *ITE*-trees are merged (Velev, 2004d).

The benefits from merging *ITE*-trees with their leaves include fewer variables and clauses and thus reduced solution space, smaller CNF file sizes, and fewer L2-cache misses; reduced Boolean Constraint Propagation, due to the eliminated intermediate variables for outputs of *ITE*s inside a tree; automatic use of signal unobservability—all clauses for a path in an *ITE*-tree become satisfied when an *ITE*-controlling signal selects another path; and guiding the SAT-solver branching and learning—each path in an *ITE*-tree is due to a different symbolic-execution trace, so that we point the SAT solver toward processing one symbolic-execution trace at a time and make it easier for the SAT solver to prune infeasible paths. If Ackermann constraints are used to eliminate the UFs and UPs, as in Barrett et al. (2002), Goel et al. (1999), Pnueli et al. (2002), Tveretina and Zantema (2003), and Zantema and Groote (2003), the resulting Boolean formulas will have fewer or no *ITE*-tees and so will benefit less from this optimisation. The speedup from merging *ITE*-trees is up to two orders of magnitude, when formally verifying complex processors (Velev, 2004d).

Figure 7 Merging an *ITE*-tree with one level of its AND/OR leaves that have a fanout count of 1. Each *ITE*-tree is represented as the conjunction of all clauses for paths from leaves to the tree output



Path from leaf G_1 to o:
$(\neg b_1 \vee \neg b_2 \vee \neg c_1 \vee o) \wedge (b_1 \vee \neg c_1 \vee \neg o) \wedge (b_2 \vee \neg c_1 \vee \neg o)$

Path from a_2 to o:
$(\neg a_2 \vee c_1 \vee \neg c_2 \vee \neg c_3 \vee o) \wedge (a_2 \vee c_1 \vee \neg c_2 \vee \neg c_3 \vee \neg o)$

Path from a_3 to o:
$(\neg a_3 \vee c_1 \vee \neg c_2 \vee c_3 \vee o) \wedge (a_3 \vee c_1 \vee \neg c_2 \vee c_3 \vee \neg o)$

Path from leaf G_2 to o:
$(\neg d_1 \vee c_1 \vee c_2 \vee o) \wedge (\neg d_2 \vee c_1 \vee c_2 \vee o) \wedge (d_1 \vee d_1 \vee c_1 \vee c_2 \vee \neg o)$

7 Summary of results

Experiments were conducted 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 physical memory, and running Red Hat Linux 9.0. Our tool flow, consisting of TLSim and EVC, was combined with the SAT solvers

siege_v4 (Ryan, 2003), and BerkMin621 (Goldberg and Novikov, 2003). The e_{ij} encoding (Goel et al., 1999) was used for g-equations, since it was found to outperform the encodings from Pnueli et al. (2002), and Velev (2003a). The Boolean correctness formulas were translated to CNF using the method described in Section 6.3.

The experiments were to formally verify safety of the benchmarks: `1dlx_c`, a single-issue five-stage pipelined DLX (Hennessy and Patterson, 2002), modelled as described in Velev and Bryant (1999b); `2dlx_ca`, a dual-issue superscalar DLX, with one complete and one ALU pipeline (Velev and Bryant, 1999b); `2dlx_cc_mc_ex_bp`, a dual-issue superscalar DLX, with two complete pipelines, exceptions, branch prediction and multicycle functional units (Velev and Bryant, 2000); `ooo_engine6`, an out-of-order processor with a completely implemented six-entry reorder buffer, completely implemented and instantiated six reservation stations, register renaming, and ALU instructions—this processor was modelled and formally verified as described in Velev (2004a), and is based on the description of the PowerPC 750 (IBM Corporation, 1999), an embedded processor that is compatible with the PowerPC ISA and has six reorder buffer entries and six reservation stations—in contrast to other out-of-order models (Hosabettu et al., 1999; Jhala and McMillan, 2002; Lahiri and Bryant, 2003), `ooo_engine6` has a completely implemented reorder buffer, and completely implemented and instantiated reservation stations; and `9vliw_bp_mc_ex_9stages_iq5`, a nine-stage, nine-wide VLIW processor that imitates the Intel Itanium (Intel Corporation, 1999; Sharangpani and Arora, 2000) in features such as predicated execution, register remapping, advanced loads, branch prediction, and multicycle functional units, exceptions, and a five-entry instruction queue (a simpler version of this processor with fewer pipeline stages and no instruction queue was formally verified in Velev (2000), and Velev and Bryant (2003)). The abstraction function was computed by controlled flushing (Burch, 1996), where the user provides a stalling schedule to override the processor stall signals, thus eliminating the ambiguity of the instruction flow during flushing and producing a simpler EUFM correctness formula.

The benchmark `1dlx_c` was formally verified in a total of 0.06 seconds; `2dlx_ca` in 0.2 seconds; `2dlx_cc_mc_ex_bp` in 0.9 seconds; `ooo_engine6` in four hours; and `9vliw_bp_mc_ex_9stages_iq5` in eight hours and nine minutes. The SAT solver siege_v4 was used for the first four benchmarks, since it was faster than BerkMin621 on their CNF formulas, while BerkMin621 had advantage for the last benchmark and was used for it.

Without Positive Equality—using the e_{ij} encoding for all equations, including p-equations between p-terms—the formal verification of `2dlx_cc_mc_ex_bp` did not complete in 90,000 seconds. Hence, Positive Equality results in at least five orders of magnitude speedup for realistic dual-issue superscalar processors. Furthermore, the speedup is increasing with the complexity of the implementation.

8 Related work

Before the use of Positive Equality and other optimisations to translate EUFM formulas to SAT, the most widely used method for formal verification of pipelined processors was theorem proving. However, the formal verification of a five-stage pipelined DLX or ARM—comparable to `1dlx_c` from Section 7—required extensive manual work by experts and often long CPU times (Börger and Mazzanti, 1997; Cyrluk, 1996; Fox, 2002; Hosabettu et al., 1998; Huggins and Van Campenhout, 1998; Jacobi and Kröning, 2000; Kröning and Paul, 2001; Müller and Paul, 2000; Tahar and Kumar, 1998; Windley, 1995). Even three-stage pipelines, executing only ALU instructions, took significant manual intervention to formally verify with theorem proving (Manolios, 2000; Sawada, 2000), or with assume-guarantee reasoning (Henzinger et al., 1998; Henzinger et al., 2000). Symbolic Trajectory Evaluation (Intel Corporation, 2000; Jain et al., 1996; Nelson et al., 1997; Seger and Bryant, 1995) also required extensive manual work to prove the correctness of just a register-immediate OR instruction in a bit-level five-stage ARM processor (Patankar et al., 1999). Other researchers had to limit the data values to four bits, the register file to one register, and the ISA to 16 instructions, to symbolically verify a bit-level pipelined processor (Bhagwati and Devadas, 1994). Various symbolic tools required long CPU time when formally verifying a pipelined DLX (Hinrichsen et al., 1999; Ritter et al., 1999), or ran out of memory (Isles et al., 1998). Custom-tailored, manually defined rewriting rules were used to formally verify a five-stage DLX (Levitt and Olukotun, 1997), and similar four-stage processors (Harman, 2001; Lis, 2000; Matthews and Launchbury, 1999), but would require modifications to work on designs described in a different coding style and significant extensions to scale for dual-issue superscalar processors. Other researchers proved only few properties of a pipelined DLX (Ivanov, 2002; Ramesh and Bhaduri, 1999), or did not present completeness argument (Mishra and Dutt, 2002)—that the properties proved will ensure correctness under all possible scenarios.

Historically, the inductive correctness criterion in Figure 1 dates back to Milner (1971), and Hoare (1972), who used it to formally verify programs by manually defining an abstraction function to map the state of an implementation program to the state of a specification program. Srivas and Bickford (1990) were first to formally verify a pipelined processor by using a theorem-proving approach and also manually defined abstraction function. Burch and Dill (1994) proposed flushing as a way to automatically compute an abstraction function and were first to formally verify a pipelined DLX. However, they had to manually provide a case-splitting expression for the conditions when the processor will fetch and complete a new instruction. Burch (1996) applied the same method to a dual-issue superscalar DLX, but had to manually define 28 case-splitting expressions and to decompose the safety correctness criterion. That decomposition was subtle enough

to warrant publication of its correctness proof as a separate paper (Windley and Burch, 1996). Hosabettu et al. (1998) used theorem proving to formally verify a single-issue pipelined DLX and a dual-issue superscalar DLX, but reported one month of manual work for each.

Our tool flow was used to formally verify a model of the Intel XScale processor with a scoreboard, specialized execution pipelined, and imprecise exceptions (Srinivasan and Velev, 2003). The tool flow was applied to formally verify a version of the M•CORE processor at Motorola, and detected two bugs in the forwarding logic, one bug in the issue logic, and corner cases that were not fully implemented (Lahiri et al., 2001). The tool flow was also used in two editions of an advanced computer architecture course (Velev, 2005a; 2003b), where undergraduate and graduate students without prior knowledge of formal methods designed and formally verified single-issue pipelined DLX processors, as well as extensions with exceptions and branch prediction, and dual-issue superscalar implementations.

Our tool flow owes much of its efficiency to the tremendous improvements in the speed of SAT solvers (Moskewicz et al., 2001; Goldberg and Novikov, 2002; Ryan, 2003); however, as determined in Section 7, even the most efficient SAT solvers would not scale for CNF formulas from complex processors, if the property of Positive Equality is not used. For comparative studies of SAT solvers, the reader is referred to Le Berre and Simon (2005), and Velev and Bryant (2003), and for surveys of recent advances in SAT to Biere and Kunz (2002), Kautz and Selman (2003), and Zhang and Malik (2002).

In the decision procedure EVC, the translation to CNF is done in a single step, by including all constraints for transitivity of equality and for functional consistency of uninterpreted functions and uninterpreted predicates, i.e., the translation is *eager*, as is also the case in Bryant et al. (2002), Lahiri and Bryant (2003), Pnueli et al. (2002), and Seshia et al. (2003). In *lazy translation* to SAT (Audemard et al., 2002; Barrett et al., 2002; de Moura et al., 2002)—constraints are added incrementally to prevent recurrence of false counterexamples—this significantly degrades the performance when deciding complex EUFM formulas (Seshia et al., 2003). Heuristics that sped up SVC (Jones et al., 1995; Barrett et al., 1996; http://sprout.Stanford.EDU/SVC), an EUFM decision procedure based on Burch and Dill's work (1994), are presented in Jones et al. (1995), and Levitt and Olukotun (1997), but did not scale for big formulas or for formulas with different structure (Barrett et al., 2002). Recent decision procedures (Bryant et al., 2002; Seshia et al., 2003)—extending EUFM with counter arithmetic, lambda expressions, and inequalities—exploit most of the optimizations in EVC, including Positive Equality.

Nested *ITE*s were first used to eliminate uninterpreted functions and uninterpreted predicates in Velev and Bryant (1998c), where bit-level functional units were abstracted with read-only instances of an Efficient Memory Model (Velev and Bryant, 1998a; Velev and Bryant, 1998b)—developed for use in symbolic simulation as a behavioural abstraction of memories, and later adopted in verification tools by Innologic Systems (Hasteer, 1999), and Synopsys (Kölbl et al., 2002).

When abstracting functional units and memories, we assume that their bit-level implementations are formally verified separately. The technology for this is already used widely in industry (Pandey and Bryant, 1999; Chen and Bryant, 2001; Jones, 2002; Parthasarathy et al., 2002).

Our tool flow is best suited for formal verification of embedded processors, including out-of-order designs such as `ooo_engine6` (see Section 7). The reader is referred to Hosabettu et al. (1999), Jhala and McMillan (2002), Lahiri and Bryant (2003), and Sawada and Hunt (2002) for techniques for formal verification of more complex out-of-order models. Note that by formally verifying an implementation processor, we prove the logical correctness of the design. However, fabrication defects may still lead to bugs in specific chips and can only be detected by testing methods (Albin, 2001; Bentley, 2001).

9 Conclusions

We presented a tool flow for high-level design and formal verification of embedded processors. The tool flow consists of:

- the term-level symbolic simulator TLSim, which accepts implementation and specification processors in the high-level hardware description language AbsHDL, as well as a simulation command file and produces an EUFM formula for correctness of the implementation;
- the decision procedure EVC that exploits Positive Equality and other optimisations to translate the EUFM correctness formula to an equivalent Boolean formula; and
- any efficient SAT procedure to prove that the Boolean correctness formula is a tautology.

Positive Equality resulted in at least five orders of magnitude speedup for realistic dual-issue superscalar processors; the speedup increases with the complexity of the implementation processor. An efficient translation to CNF led to another two orders of magnitude speedup. The tool flow was used at Motorola to formally verify a model of the M•CORE processor and detected bugs. TLSim and EVC are available from Velev (2004e).

Future work will extend EVC with the capability to produce proofs for every formula that it reports valid, e.g., as done in Stump (2002). We will also develop a tool for automatic translation of a formally verified high-level microprocessor model to a description in a synthesisable bit-level HDL.

References

Aagaard, M.D., Cook, B., Day, N.A. and Jones, R.B. (2003) 'A framework for superscalar microprocessor correctness statements', *Software Tools for Technology Transfer (STTT)*, May, Vol. 4, No. 3, pp.298–312.

Aagaard, M.D., Day, N.A. and Lou, M. (2002) 'Relating multi-step and single-step microprocessor correctness statements', in Aagaard, M.D. and O'Leary, J.W. (Eds.): *Formal Methods in Computer-Aided Design (FMCAD '02)*, *LNCS 2517*, November, Springer-Verlag, pp.123–141.

Ackermann, W. (1954) *Solvable Cases of the Decision Problem*, North-Holland, Amsterdam.

Albin, K. (2001) 'Nuts and bolts of core and SoC verification', *38th Design Automation Conference (DAC '01)*, June, Las Vegas, NV, USA, pp.249–252.

Audemard, G., Bertoli, P., Cimatti, A., Korniowicz, A. and Sebastiani, R. (2002) 'A SAT based approach for solving formulas over Boolean and linear mathematical propositions', *11th International Conference on Automated Deduction (CADE '02)*, *LNCS 2392,* July, Springer-Verlag, pp.195–210.

Barrett, C., Dill, D. and Stump, A. (2002) 'Checking satisfiability of first-order formulas by incremental translation to SAT', *Computer-Aided Verification (CAV '02)*, *LNCS 2404*, July, Springer-Verlag, pp.187–201.

Barrett, C.W., Dill, D.L. and Levitt, J.R. (1996) 'Validity checking for combinations of theories with equality', Srivas, M. and Camilleri, A. (Eds.): *Formal Methods in Computer-Aided Design (FMCAD '96*), *LNCS 1166*, November, Springer-Verlag, pp.187–201.

Bentley, B. (2001) 'Validating the Intel® Pentium® 4 microprocessor', *38th Design Automation Conference (DAC '01)*, June, Las Vegas, NV, USA, pp.244–248.

Bhagwati, V. and Devadas, S. (1994) 'Automatic verification of pipelined microprocessors', *31st Design Automation Conference (DAC '94)*, June, San Diego, California, USA, pp.603–608.

Biere, A. and Kunz, W. (2002) 'SAT and ATPG: Boolean engines for formal hardware verification', *International Conference on Computer Aided Design (ICCAD '02)*, November, San Jose, California, USA, pp.782–785.

Blank, C., Eveking, H., Levihn, J. and Ritter, G. (2001) 'Symbolic simulation techniques – state-of-the-art and applications', *IEEE International High-Level Design Validation and Test Workshop (HLDVT '01)*, December, pp.45–50.

Börger, E. and Mazzanti, S. (1997) 'A practical method for rigorously controllable hardware design', in Bowen, J., Hinchey, M. and Till, D. (Eds.): *10th International Conference of Z Users (ZUM '97)*, *LNCS 1212*, April, Springer-Verlag, pp.151–187.

Brglez, F. and Fujiwara, H. (1985) 'A neutral netlist of 10 combinational benchmark circuits', *International Symposium on Circuits and Systems (ISCAS '85)*, June, Kyoto, Japan, pp.785–794.

Bryant, R.E. (1986) 'Graph-based algorithms for Boolean function manipulation', *IEEE Transactions on Computers*, August, Vol. C-35, No. 8, pp.677–691.

Bryant, R.E. (1992) 'Symbolic Boolean manipulation with ordered binary-decision diagrams', *ACM Computing Surveys*, September, Vol. 24, No. 3, pp.293–318.

Bryant, R.E. and Meinel, C. (2001) 'Ordered binary decision diagrams', in Hassoun, S. and Sasao, T. (Eds.): *Logic Synthesis and Verification*, Kluwer Academic Publishers, Boston/Dordrecht/London.

Bryant, R.E. and Velev, M.N. (2002) 'Boolean satisfiability with transitivity constraints', *ACM Transactions on Computational Logic (TOCL)*, October, Vol. 3, No. 4, pp.604–627.

Bryant, R.E., German, S. and Velev, M.N. (2001) 'Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic', *ACM Transactions on Computational Logic (TOCL)*, January, Vol. 2, No. 1, pp.93–134.

Bryant, R.E., Lahiri, S.K. and Seshia, S.A. (2002) 'Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions', in Brinksma, E. and Larsen, K.G. (Eds.): *Computer-Aided Verification (CAV'02)*, *LNCS 2404*, July, Springer-Verlag, pp.78–92.

Burch, J.R. (1996) 'Techniques for verifying superscalar microprocessors', *33rd Design Automation Conference (DAC '96)*, June, Las Vegas, Nevada, USA, pp.552–557.

Burch, J.R. and Dill, D.L. (1994) 'Automated verification of pipelined microprocessor control', in Dill, D.L. (Ed.): *Computer-Aided Verification* (*CAV '94)*, *LNCS 818*, June, Springer-Verlag, pp.68–80.

Chen, Y-A. and Bryant, R.E. (2001) 'An efficient graph representation for arithmetic circuit verification', *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, December, Vol. 20, No. 12, pp.1442–1454.

Cyrluk, D. (1996) 'Inverting the abstraction mapping: a methodology for hardware verification', in Srivas, M. and Camilleri, A. (Eds.): *Formal Methods in Computer-Aided Design (FMCAD '96)*, *LNCS 1166*, November, Springer-Verlag, pp.172–186.

de Moura, L., Rueß, H. and Sorea, M. (2002) 'Lazy theorem proving for bounded model checking over infinite domains', *11th International Conference on Automated Deduction (CADE '02)*, *LNCS 2392*, July, Springer-Verlag, pp.438–455.

Filliâtre, J.C., Owre, S., Rueß, H. and Shankar, N. (2001) 'ICS: integrated canonizer and solver', in Berry, G., Comon, H. and Finkel, A. (Eds.): *Computer-Aided Verification (CAV '01)*, LNCS 2102, July, Springer-Verlag, pp.246–249.

Fox, A. (2002) *Formal verification of the ARM6 micro-architecture*, Technical Report UCAM-CL-TR-548, November, Computer Laboratory, University of Cambridge.

Goel, A., Sajid, K., Zhou, H., Aziz, A. and Singhal, V. (1998) 'BDD based procedures for a theory of equality with uninterpreted functions', in Hu, A.J. and Vardi, M.Y. (Eds.): *Computer-Aided Verification (CAV '98)*, LNCS 1427, June Springer-Verlag, pp.244–255.

Goldberg, E. and Novikov, Y. (2002) 'BerkMin: a fast and robust SAT solver', *Design, Automation, and Test in Europe (DATE '02)*, March, pp.142–149.

Goldberg, E. and Novikov, Y. (2003) *Personal Communication*, June.

Grove, A.S. (1999) *Only the Paranoid Survive: How to Exploit the Crisis Points That Challenge Every Company*, Currency, New York.

Harman, N.A. (2001) 'Verifying a simple pipelined microprocessor using Maude', in Cerioli, M. and Reggio, G. (Eds.): *15th International Workshop on Recent Trends in Algebraic Development Techniques (WADT '01)*, *LNCS 2267*, April, Springer-Verlag, pp.128–151.

Hasteer, G. (1999) *Personal Communication*, February.

Hennessy, J.L. and Patterson, D.A. (2002) *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann Publishers, San Francisco, CA.

Henzinger, T.A., Qadeer, S. and Rajamani, S.K. (1998) 'You assume, we guarantee: methodology and case studies', in Hu, A.J. and Vardi, M.Y. (Eds.): *Computer-Aided Verification (CAV '98)*, *LNCS 1427*, June, Springer-Verlag, pp.440–451.

Henzinger, T.A., Qadeer, S. and Rajamani, S.K. (2000) 'Decomposing refinement proofs using assume-guarantee reasoning', *International Conference on Computer-Aided Design*, November, San Jose, California, USA.

Hinrichsen, H., Eveking, H. and Ritter, G. (1999) 'Formal synthesis for pipeline design', in Calude, C.S. and Dinneen, M.J. (Eds.): *2nd International Conference on Discrete Mathematics and Theoretical Computer Science (DMTCS '99) and the 5th Australasian Theory Symposium (CATS '99)*, *Australian Computer Science Communications*, Springer-Verlag, Auckland, New Zealand, Vol. 21, No. 3.

Hoare, C.A.R. (1972) 'Proof of correctness of data representations', *Acta Informatica*, Vol. 1, pp.271–281.

Hosabettu, R., Srivas, M. and Gopalakrishnan, G. (1998) 'Decomposing the proof of correctness of pipelined microprocessors', in Hu, A.J. and Vardi, M.Y. (Eds.): *Computer-Aided Verification (CAV '98)*, *LNCS 1427,* June, Springer-Verlag, pp.122–134.

Hosabettu, R., Srivas, M. and Gopalakrishnan, G. (1999) 'Proof of correctness of a processor with reorder buffer using the completion functions approach', in Halbwachs, N. and Peled, D. (Eds.): *Computer-Aided Verification (CAV '99)*, *LNCS 1633*, July, Springer-Verlag, pp.47–59.

Huggins, J.K. and Van Campenhout, D. (1998) 'Specification and verification of pipelining in the ARM2 RISC microprocessor', *ACM Transactions on Design Automation of Electronic Systems*, October, Vol. 3, No. 4, pp.563–580.

Hulgaard, H., Williams, P.F. and Andersen, H.R. (1999) 'Equivalence checking of combinational circuits using Boolean expression diagrams', *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, Vol. 18, No. 7, July, pp.903–917.

IBM Corporation (1999) *PowerPC 740™/PowerPC 750™: RISC Microprocessor User's Manual*.

Intel Corporation (1999) *IA-64 Application Developer's Architecture Guide*, May, http://developer.intel.com/ design/ia-64/architecture.htm

Intel Corporation (2000) *Partial Bibliography of STE Related Research*, http://intel.com/research/scl/library STE_Bibliography.pdf.

Isles, A.J., Hojati, R. and Brayton, R.K. (1998) 'Computing reachable control states of systems modeled with uninterpreted functions and infinite memory', in Hu, A.J. and Vardi, M.Y. (Eds.): *Computer-Aided Verification (CAV '98)*, *LNCS 1427,* Springer-Verlag, June, pp.256–267.

Ivanov, L. (2002) 'Modeling and verification of a pipelined CPU', *Midwest Symposium on Circuits and Systems (MWSCAS '02)*, August.

Jacobi, C. and Kröning, D. (2000) 'Proving the correctness of a complete microprocessor', *30. Jahrestagung der Gesellshaft für Informatik*, Springer-Verlag.

Jain, A., Nelson, K.A. and Bryant, R.E. (1996) 'Verifying nondeterministic implementations of deterministic systems', in Srivas, M. and Camilleri, A. (Eds.): *Formal Methods in Computer-Aided Design (FMCAD '96)*, *LNCS 1166*, November, Springer-Verlag, pp.109–125.

Jhala, R. and McMillan, K.L. (2002) 'Microarchitecture verification by compositional model checking', in Berry, G., Comon, H. and Finkel, A. (Eds.): *Computer-Aided Verification (CAV '01)*, *LNCS 2404*, July, Springer-Verlag, pp.526–538.

Johnson, D.S. and Trick, M.A. (Eds.) (1993) *The Second DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, http://dimacs.rutgers.edu/challenges.

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

Jones, R.B., Dill, D.L. and Burch, J.R. (1995) 'Efficient validity checking for processor verification', *International Conference on Computer-Aided Design (ICCAD '95)*, November, San Jose, California, USA, pp.2–6.

Kautz, H. and Selman, B. (2003) 'Ten challenges *redux*: recent progress in propositional reasoning and search', in Rossi, F. (Ed.): *Principles and Practice of Constraint Programming (CP '03)*, *LNCS 2833*, September–October, Springer-Verlag.

Keutzer, K., Malik, S., Newton, A.R., Rabaey, J.M. and Sangiovanni-Vincentelli, A. (2000) 'System-level design: orthogonalization of concerns and platform-based design', *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, December, Vol. 19, No. 12, pp.1523–1543.

Kölbl, A., Kukula, J.H., Antreich, K. and Damiano, R.F. (2002) 'Handling special constructs in symbolic simulation', *39th Design Automation Conference (DAC '02)*, June, New Orleans, LA, USA, pp.105–110.

Kröning, D. and Paul, W.J. (2001) 'Automated pipeline design', *38th Design Automation Conference (DAC '01)*, June, Las Vegas, NV, USA, pp.810–815.

Lahiri, S., Pixley, C. and Albin, K. (2001) '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, Monterey, California, USA, pp.109–114.

Lahiri, S.K. and Bryant, R.E. (2003) 'Deductive verification of advanced out-of-order microprocessors', *Computer-Aided Verification (CAV '03)*, *LNCS*, July, Springer-Verlag, Monterey, California, USA.

Larrabee, T. (1992) 'Test pattern generation using Boolean satisfiability', *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, January, Vol. 11, No. 1, pp.4–15.

Le Berre, D. and Simon, L. (2005) 'The SAT 2005 competition: fourth edition', *8th International Conference on Theory and Applications of Satisfiability Testing (SAT '05)*, June, http:// www.lri.fr/~simon/contest/results/.

Levitt, J. and Olukotun, K. (1997) 'Verifying correct pipeline implementation for microprocessors', *International Conference on Computer-Aided Design (ICCAD '97)*, November, San Jose, CA, USA, pp.162–169.

Lis, M.N. (2000) *Superscalar Processors via Automatic Microarchitecture Transformations*, M.S. thesis, June, Department of Electrical Engineering and Computer Science, M.I.T.

Manolios, P. (2000) 'Correctness of pipelined machines', in Hunt Jr., W.A., and Johnson, S.D. (Eds.): *Formal Methods in Computer-Aided Design (FMCAD '00)*, *LNCS 1954*, November, Springer-Verlag, pp.161–178.

Marques-Silva, J.P. and e Silva, L.G. (1999) 'Algorithms for satisfiability in combinational circuits based on backtrack search and recursive learning', *12th Symposium on Integrated Circuits and Systems Design (SBCCI '99)*, September–October, Natal, Brazil, pp.192–195.

Matthews, J. and Launchbury, J. (1999) 'Elementary microarchitecture algebra', in Halbwachs, N. and Peled, D. (Eds.): *Computer-Aided Verification (CAV '99)*, *LNCS 1633*, Springer-Verlag, June, pp.288–300.

Milner, R. (1971) 'An algebraic definition of simulation between programs', *2nd International Joint Conference on Artificial Intelligence*, The British Computer Society, pp.481–489.

Mishra, P. and Dutt, N. (2002) 'Modeling and verification of pipelined embedded processors in the presence of hazards and exceptions', *IFIP WCC 2002 Stream 7 on Distributed and Parallel Embedded Systems (DIPES '02)*, August, Montréal, Québec, Canada, pp.81–90.

Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L. and Malik, S. (2001) 'Chaff: engineering an efficient SAT solver', *38th Design Automation Conference (DAC '01)*, June, Las Vegas, NV, USA, pp.530–535.

Müller, S.M. and Paul, W.J. (2000) *Computer Architecture: Complexity and Correctness*, Springer-Verlag.

Nelson, K.L., Jain, A. and Bryant, R.E. (1997) 'Formal verification of a superscalar execution unit', *34th Design Automation Conference (DAC '97)*, June, Anaheim, California, USA, pp.161–166.

Pandey, M. and Bryant, R.E. (1999) 'Exploiting symmetry when verifying transistor-level circuits by symbolic trajectory evaluation', *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, July, Vol. 18, No. 7, pp.918–935.

Parthasarathy, G., Iyer, M.K., Feng, T., Wang, L-C., Cheng, K-T., and Abadir, M.S. (2002) 'Combining ATPG and symbolic simulation for efficient validation of embedded array systems', *International Test Conference (ITC '02)*, October, Baltimore, MD, pp.203–212.

Pasztor, A. and Landers, P. (1999) 'Toshiba to pay $2B settlement on laptops', *ZDNet News*, November, http://www.zdnet.com/zdnn/stories/news/0,4586,2385037,00.html

Patankar, V.A., Jain, A. and Bryant, R.E. (1999) 'Formal verification of an ARM processor', *12th International Conference on VLSI Design*, January, Goa, India, pp.282–287.

Plaisted, D.A. and Greenbaum, S. (1985) 'A structure preserving clause form translation', *Journal of Symbolic Computation (JSC)*, Vol. 2, pp.293–304.

Pnueli, A., Rodeh, Y., Strichman, O. and Siegel, M. (2002) 'The small model property: how small can it be?', *Journal of Information and Computation*, October, Vol. 178, No. 1, pp.279–293.

Ramesh, S. and Bhaduri, P. (1999) 'Validation of pipelined processor designs using Esterel tools: a case study', in Halbwachs, N. and Peled, D. (Eds.): *Computer-Aided Verification (CAV '99)*, *LNCS 1633*, July, Springer-Verlag, pp.84–95.

Ritter, G., Eveking, H. and Hinrichsen, H. (1999) 'Formal verification of designs with complex control by symbolic simulation', in Pierre, L. and Kropf, T. (Eds.): *Correct Hardware Design and Verification Methods (CHARME '99)*, *LNCS 1703*, September, Springer-Verlag, pp.234–249.

Ryan, L. (2003) *Siege SAT Solver v.4*, http://www.cs.sfu.ca/~loryan/personal/.

Sawada, J. (2000) 'Verification of a simple pipelined machine', *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Publishers, Boston/Dordrecht/London, pp.137–150.

Sawada, J. and Hunt Jr., W.A. (2002) 'Verification of FM9801: out-of-order processor with speculative execution and exceptions that may execute self-modifying code', *Journal on Formal Methods in System Design (FMSD), special issue on Microprocessor Verifications*, March, Vol. 20, No. 2, pp.187–222.

Seger, C-J.H. and Bryant, R.E. (1995) 'Formal verification by symbolic evaluation of partially-ordered trajectories', *Formal Methods in System Design (FMSD)*, March, Vol. 6, No. 2, pp.147–190.

Seshia, S.A., Lahiri, S.K. and Bryant, R.E. (2003) 'A hybrid SAT-based decision procedure for separation logic with uninterpreted functions', *40th Design Automation Conference (DAC '03)*, June, Anaheim, CA, USA, pp.425–430.

Sharangpani, H. and Arora, K. (2000) 'Itanium processor microarchitecture', *IEEE Micro*, September–October, Vol. 20, No. 5, pp.24–43.

Sheeran, M. and Stålmarck, G. (2000) 'A tutorial on Stålmarck's proof procedure for propositional logic', *Formal Methods in System Design (FMSD)*, January, Vol. 16, No. 1, pp.23–58.

Somenzi, F. (1999) *CUDD version 2.3.0*, http://vlsi.colorado.edu/~fabio

Somenzi, F. (2001) 'Efficient manipulation of decision diagrams', *International Journal on Software Tools for Technology Transfer (STTT)*, Vol. 3, No. 2, pp.171–181.

Srinivasan, S.K. and Velev, M.N. (2003) 'Formal verification of an Intel XScale processor model with scoreboarding, specialized execution pipelines, and imprecise data-memory exceptions', *Formal Methods and Models for Codesign (MEMOCODE '03)*, June, Mont Saint-Michel, France, pp.65–74.

Srivas, M. and Bickford, M. (1990) 'Formal verification of a pipelined microprocessor', *IEEE Software*, September–October, Vol. 7, No. 5, pp.52–64.

Stålmarck, G. (1989) *A System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from a Formula*, Swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 897 (1994), European Patent No. 0403 454 (1995).

Stump, A. (2002) *Checking Validity of Proofs with CVC and Flea*, Ph.D. thesis, Department of Computer Science, Stanford University, September.

Tahar, S. and Kumar, R. (1998) 'A practical methodology for the formal verification of RISC processors', *Formal Methods in Systems Design (FMSD)*, September, Vol. 13, No. 2, pp.159–225.

Tennenhouse, D. (2000) 'Proactive computing', *Communications of the ACM*, May, Vol. 43, No. 5, pp.43–50.

Tseitin, G.S. (1968) 'On the complexity of derivation in propositional calculus', in *Studies in Constructive Mathematics and Mathematical Logic*, Part 2, pp.115–125, Reprinted in Siekmann, J. and Wrightson, G. (Eds.): *Automation of Reasoning*, Vol. 2, Springer-Verlag, 1983, pp.466–483.

Tveretina, O. and Zantema, H. (2003) *A Proof System and a Decision Procedure for Equality Logic*, Technical Report, Department of Computer Science, Technical University of Eindhoven, http://www.win.tue.nl/~hzantema/TZ.pdf.

Velev, M.N. (2000) 'Formal verification of VLIW microprocessors with speculative execution', in Emerson, E.A. and Sistla, A.P., (Eds.): *Computer-Aided Verification (CAV '00)*, *LNCS 1855*, July, Springer-Verlag, pp.296–311.

Velev, M.N. (2001) 'Automatic abstraction of memories in the formal verification of superscalar microprocessors', in Margaria, T. and Yi, W. (Eds.): *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, *LNCS 2031*, April, Springer-Verlag, pp.252–267.

Velev, M.N. (2003a) 'Automatic abstraction of equations in a logic of equality', in Mayer, M.C. and Pirri, F. (Eds.): *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX '03)*, *LNAI 2796*, September, Springer-Verlag, pp.196–213.

Velev, M.N. (2003b) 'Collection of high-level microprocessor bugs from formal verification of pipelined and superscalar designs', *International Test Conference (ITC '03)*, October, Charlotte, NC, USA, pp.138–147.

Velev, M.N. (2004a) 'Using automatic case splits and efficient CNF translation to guide a SAT solver when formally verifying out-of-order processors', *Artificial Intelligence and Mathematics (AI&MATH '04)*, January, Fort Lauderdale, Florida, USA, pp.242–254.

Velev, M.N. (2004b) 'Using positive equality to prove liveness for pipelined microprocessors', *Asia and South Pacific Design Automation Conference (ASP-DAC '04)*, January, Yokohama, Japan, pp.316–321.

Velev, M.N. (2004c) 'Efficient translation of Boolean formulas to CNF in formal verification of microprocessors', *Asia and South Pacific Design Automation Conference (ASP-DAC '04)*, January, Yokohama, Japan, pp.310–315.

Velev, M.N. (2004d) 'Exploiting signal unobservability for efficient translation to CNF in formal verification of microprocessors', *Design, Automation and Test in Europe (DATE '04)*, February, Paris, France, pp.266–271.

Velev, M.N. (2004e) TLSim and EVC, http://www.ece.cmu.edu/~mvelev.

Velev, M.N. (2005a) 'Integrating formal verification into an advanced computer architecture course', *IEEE Transactions on Education*, May, Vol. 48, No. 2, pp.216–222.

Velev, M.N. (2005b) 'Automatic formal verification of liveness for pipelined processors with multicycle functional units', in Borrione, D. and Paul, W.J. (Eds.): *13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '05)*, LNCS 3725, Springer-Verlag, October, pp.97–113.

Velev, M.N. and Bryant, R.E. (1998a) 'Efficient modeling of memory arrays in symbolic ternary simulation', in Steffen, B., (Ed.): *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, *LNCS 1384*, March–April, Springer-Verlag, pp.136–150.

Velev, M.N. and Bryant, R.E. (1998b) 'Incorporating timing constraints in the efficient memory model for symbolic ternary simulation', *International Conference on Computer Design (ICCD '98)*, October, Austin, Texas, USA, pp.400–406.

Velev, M.N. and Bryant, R.E. (1998c) 'Bit-level abstraction in the verification of pipelined microprocessors by correspondence checking', in Gopalakrishnan, G. and Windley, P. (Eds.): *Formal Methods in Computer-Aided Design (FMCAD '98)*, *LNCS 1522*, November, Springer-Verlag, pp.18–35.

Velev, M.N. and Bryant, R.E. (1999a) 'Exploiting positive equality and partial non-consistency in the formal verification of pipelined microprocessors', *36th Design Automation Conference (DAC '99)*, June, New Orleans, LA, USA, pp.397–401.

Velev, M.N. and Bryant, R.E. (1999b) 'Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic', in Pierre, L. and Kropf, T. (Eds.): *Correct Hardware Design and Verification Methods (CHARME '99)*, LNCS 1703, September, Springer-Verlag, pp.37–53.

Velev, M.N. and Bryant, R.E. (2000) 'Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction', *37th Design Automation Conference (DAC '00)*, June, Los Angeles, CA, USA, pp.112–117.

Velev, M.N. and Bryant, R.E. (2001) 'EVC: a validity checker for the logic of equality with uninterpreted functions and memories, exploiting positive equality and conservative transformations', in Berry, G., Comon, H. and Finkel, A. (Eds.): *Computer-Aided Verification (CAV '01)*, *LNCS 2102*, July, Springer-Verlag, pp.235–240.

Velev, M.N. and Bryant, R.E. (2003) 'Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors', *Journal of Symbolic Computation (JSC)*, February, Vol. 35, No. 2, pp.73–106.

Williams, P.F. (2000) *Formal Verification Based on Boolean Expression Diagrams*, Ph.D. thesis, Department of Information Technology, Technical University of Denmark, Lyngby, Denmark.

Windley, P.J. (1995) 'Verifying pipelined microprocessors', *Conference on Hardware Description Languages (CHDL '95)*, August, Tokyo, Japan, pp.503–511.

Windley, P.J. and Burch, J.R. (1996) 'Mechanically checking a lemma used in an automatic verification tool', in Srivas, M. and Camilleri, A. (Eds.): *Formal Methods in Computer-Aided Design (FMCAD '96)*, *LNCS 1166*, November, Springer-Verlag, pp.362–376.

Zantema, H. and Groote, J.F. (2003) 'Transforming equality logic to propositional logic', *4th International Workshop on First Order Theorem Proving (FTP '03)*, June.

Zhang, L. and Malik, S. (2002) 'The quest for efficient Boolean satisfiability solvers', in Brinksma, E. and Larsen, K.G. (Eds.): *Computer-Aided Verification (CAV '02)*, *LNCS 2404*, July, Springer-Verlag, pp.17–36.

Zhang, L. and Malik, S. (2003) 'Cache performance of SAT solvers: a case study for efficient implementation of algorithms', *6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03)*, May, Santa Margherita Ligure, Italy, pp.287–298.