Automatic Abstraction of Memories in the Formal Verification of Superscalar Microprocessors[1]

Miroslav N. Velev
mvelev@ece.cmu.edu
http://www.ece.cmu.edu/~mvelev
Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.

Abstract. A system of conservative transformation rules is presented for abstracting memories whose forwarding logic interacts with stalling conditions for preserving the memory semantics in microprocessors with in-order execution. Microprocessor correctness is expressed in the logic of Equality with Uninterpreted Functions and Memories (EUFM) [6]. Memory reads and writes are abstracted as arbitrary uninterpreted functions in such a way that the forwarding property of the memory semantics—that a read returns the data most recently written to an equal write address—is satisfied completely only when exactly the same pair of one read and one write address is compared for equality in the stalling logic. These transformations are applied entirely automatically by a tool for formal verification of microprocessors, based on EUFM, the Burch and Dill flushing technique [6], and the properties of Positive Equality [3]. An order of magnitude reduction is achieved in the number of e_{ij} Boolean variables [9] that encode the equality comparisons of register identifiers in the correctness formulas for single-issue pipelined and dual-issue superscalar microprocessors with multicycle functional units, exceptions, and branch prediction. That results in up to 40× reduction in the CPU time for the formal verification of the dual-issue superscalar microprocessors.

1 Introduction

The motivation for this work is the complexity of the formal verification of correct microprocessors. The formal verification is done with the Burch and Dill flushing technique [6] by exploiting the properties of Positive Equality [3] in order to translate the correctness formula from the logic of Equality with Uninterpreted Functions and Memories (EUFM) to propositional logic. The translation is done by a completely automatic tool [16][21]. The resulting Boolean formula can be evaluated with either BDDs [2] or Boolean Satisfiability (SAT) checkers for being a tautology, which implies that the original EUFM correctness formula is universally valid, i.e., the processor is correct under all possible conditions.

Recently we showed that errors in complex realistic microprocessors are detected in CPU time that is up to orders of magnitude smaller than the time to prove the correctness of a bug-free version of the same design [20]. The present paper aims to speed up the verification of correct microprocessors with multicycle functional units, exceptions, and branch prediction, where reads and writes of user-visible state are not reordered and occur according to their program sequence.

1. This research was supported by the SRC under contract 00-DC-684.

T. Margaria and W. Yi (Eds.): TACAS 2001, LNCS 2031, pp. 252-267, 2001.
© Springer-Verlag Berlin Heidelberg 2001

2 Background

In this work, the logic of EUFM [6] is used for the definition of high-level models of both the implementation and the specification microprocessors. The syntax of EUFM includes terms and formulas. Terms are used in order 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 on a list of argument terms, a domain variable, or an *ITE* operator selecting between two argument terms based on a controlling formula, such that *ITE*(*formula*, $term_1$, $term_2$) will evaluate to $term_1$ when *formula* = **true** and to $term_2$ when *formula* = **false**. Formulas are used in order 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.

UFs and UPs are used to abstract away the implementation details of functional units by replacing them with "black boxes" that satisfy no particular properties other than that of *functional consistency*—the same combinations of values to the inputs of the UF (or UP) produce the same output value. Three possible ways to impose the property of functional consistency of UFs and UPs are Ackermann constraints [1], nested *ITE*s [3][16], and "pushing-to-the-leaves" [16]. In the nested *ITE*s scheme, the first application of some UF, $f(a_1, b_1)$, is replaced by a new domain variable c_1. A second application, $f(a_2, b_2)$, is replaced by $ITE((a_2 = a_1) \wedge (b_2 = b_1), c_1, c_2)$, where c_2 is a new domain variable. A third one, $f(a_3, b_3)$, is 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 domain variable, and so on.

The syntax for terms can be extended to model memories by means of the functions *read* and *write*, where *read* takes 2 argument terms serving as memory and address, respectively, while *write* takes 3 argument terms serving as memory, address, and data. Both functions return a term. Also, they can be viewed as a special class of (partially interpreted) uninterpreted functions in that they are defined to satisfy the forwarding property of the memory semantics, namely that *read*(*write*(*mem*, *aw*, *d*), *ar*) = *ITE*(*ar* = *aw*, *d*, *read*(*mem*, *ar*)), in addition to the property of functional consistency. Versions of *read* and *write* that extend the syntax for formulas can be defined similarly, such that the version of *read* will return a formula and the version of *write* will take a formula as its third argument. Both terms and formulas are called expressions.

The correctness criterion is a commutative diagram [6]. It requires that one step of the Implementation transition function followed by flushing should produce equal user-visible state as first flushing the Implementation and then using the resulting user-visible state to apply the Specification transition function between 0 and k times, where k is the issue-width of the Implementation. *Flushing* of the processor is done by feeding it with bubbles until all instructions in flight complete their execution, computing an abstraction function that maps Implementation states to a Specification state. (The difference between a bubble and a nop is that a bubble does not modify any user-visible state, while a nop increments the PC.) The correctness criterion is expressed by an EUFM formula of the form:

$$m_{1,0} \wedge m_{2,0} \dots \wedge m_{n,0} \;\vee\; m_{1,1} \wedge m_{2,1} \dots \wedge m_{n,1} \;\vee \dots \vee\; m_{1,k} \wedge m_{2,k} \dots \wedge m_{n,k}, \qquad (1)$$

where n is the number of user-visible state elements in the implementation processor, k is the maximum number of instructions that the processor can fetch in a clock cycle, and $m_{i,j}$, $1 \le i \le n$, $0 \le j \le k$, is an EUFM formula expressing the condition that user-visible state element i is updated by the first j instructions from the ones fetched in a single clock cycle. (See the electronic version of [16] for a detailed discussion.) The EUFM formulas $m_{1,j}$, $m_{2,j}$, ..., $m_{n,j}$, $0 \le j \le k$, are conjuncted in order to ensure that the user-visible state elements are updated in "sync" by the same number of instructions. The correctness criterion expresses a safety property that the processor completes between 0 and k of the newly fetched k instructions.

Positive Equality allows the identification of two types of terms in the structure of an EUFM formula—those which appear only in positive equations and are called *p-terms*, and those which can appear in both positive and negative equations and are called *g-terms* (for general terms). A *positive equation* is never negated (or appears under an even number of negations) and is not part of the controlling formula for an *ITE* operator. A *negative equation* appears under an odd number of negations or as part of the controlling formula for an *ITE* operator. The computational efficiency from exploiting Positive Equality is due to a theorem which states that the truth of an EUFM formula under a maximally diverse interpretation of the p-terms implies the truth of the formula under any interpretation. The classification of p-terms vs. g-terms is done before UFs and UPs are eliminated by nested *ITE*s, such that if an UF is classified as a p-term (g-term), the new domain variables generated for its elimination are also considered to be p-terms (g-terms). After the UFs and the UPs are eliminated, a maximally diverse interpretation is one where: the equality comparison of two syntactically identical (i.e., exactly the same) domain variables evaluates to **true**; the equality comparison of a p-term domain variable with a syntactically distinct domain variable evaluates to **false**; and the equality comparison of a g-term domain variable with a syntactically distinct g-term domain variable could evaluate to either **true** or **false** and can be encoded with a dedicated Boolean variable—an e_{ij} variable [9].

In order to fully exploit the benefits of Positive Equality, the designer of a high-level processor must use a set of suitable abstractions and conservative approximations. For example, an equality comparison of two data operands, as used to determine the condition to take a branch-on-equal instruction, must be abstracted with an UP in both the Implementation and the Specification, so that the data operand terms will not appear in negated equations but only as arguments to UPs and UFs and hence will be classified as p-terms. Similarly, a Finite State Machine (FSM) model of a memory has to be employed for abstracting the Data Memory in order for the addresses, which are produced by the ALU and also serve as data operands, to be classified as p-terms. In the FSM abstraction of a memory, the present memory state is a term that is stored in a latch. Reads are modeled with an UF f_r that depends on the present memory state and the address, while producing a term for the read data. Writes are modeled with an UF f_u that depends on the present memory state, the address, and a data term, producing a term for the new memory state, which is to be stored in the latch. The result is that data values produced by the Register File, the ALU, and the Data Memory can be classified as p-terms, while only the register identifiers, whose equations control forwarding and stalling conditions that can be negated, are classified as g-terms.

We will refer to a transformation on the implementation and specification processors as a *conservative approximation* if it omits some properties, making the new processor models more general than the original ones. Note that the same transformation is applied to both the implementation and the specification processors. However, if the more general model of the implementation is verified against the more general model of the specification, so would be the original implementation against the original specification, whose additional properties were not necessary for the verification.

Proposition 1. *The FSM model of a memory, based on uninterpreted functions* f_u *and* f_r, *is a conservative approximation of a memory.*

Proof. If a processor is proved correct with the FSM model of a memory where the update function f_u and the read function f_r are completely arbitrary uninterpreted functions that do not satisfy the forwarding property of the memory semantics, then the processor will be correct for any implementation of f_u and f_r, including $f_u \equiv write$ and $f_r \equiv read$. ❑

3 Automatic Abstraction of Memories

When abstracting memories in [19], the following transformations were applied automatically by the verification tool when processing the EUFM correctness formula, starting from the leaves of that formula:

$$read(m, a) \rightarrow f_r(m, a) \tag{2}$$

$$write(m, a, d) \rightarrow f_u(m, a, d) \tag{3}$$

$$ITE(e \wedge (ra = wa), d, f_r(m, ra)) \rightarrow f_r(ITE(e, f_u(m, wa, d), m), ra) \tag{4}$$

Transformations (2) and (3) are the same as those used in the abstraction of the Data Memory, described in Sect. 2. Transformation (4) occurs in the cases when one level of forwarding logic is used to update the data read from address *ra* of the previous state *m* for the memory, where function *read* is already abstracted with UF f_r. Accounting for the forwarding property of the memory semantics that was satisfied before function *read* was abstracted with UF f_r, the left handside of (4) is equivalent to $read(ITE(e, write(m, wa, d), m), ra)$, i.e., to a read from address *ra* of the state of memory *m* after a write to address *wa* with data *d* is done under the condition that formula *e* is **true**. On the right handside of (4), functions *read* and *write* are again abstracted with f_r and f_u after accounting for the forwarding property. Multiple levels of forwarding are abstracted by recursive applications of (4), starting from the leaves of the correctness formula. Uninterpreted functions f_u and f_r can be automatically made unique for every memory, where a memory is identified by a unique domain variable serving as the memory argument at the leaves of a memory state term. Hence, f_u and f_r will no longer be functionally consistent across memories—a conservative approximation.

After all memories were automatically abstracted as presented above, the tool in [19] checked if an address term for an abstracted memory was still used in a negated equation, i.e., was a g-term. If so, then the abstraction for that memory was undone. Hence, abstraction was performed automatically only for a memory whose addresses are p-terms outside the memory and the forwarding logic for it. From Proposition 1, it follows that such an abstraction is a conservative approximation. The condition that

address terms of abstracted memories are used only as p-terms outside the abstracted memories avoids false negatives that might result when a (negated) equation of two address terms will imply that a write to one of the addresses will (not) affect a read from the other address in the equation when that read is performed later—a property that is lost in the abstraction with UFs. In the architecture verified in [19], transformations (2) – (4) worked for the Branch-Address Register File, whose forwarding logic did not interact with stalling logic for preserving the correctness of the memory semantics for that register file, as the branch-address results were available for forwarding right after the Execute stage. However, the above abstractions were not applicable to the Integer and Floating-Point Register Files that did have stalling logic interact with their forwarding logic.

The contribution made with this paper is the idea of a *hybrid memory model*, where the forwarding property of the memory semantics is satisfied fully for only those levels of forwarding where exactly the same pair of one read and one write address is compared for equality outside the abstracted memory, i.e., in the EUFM formula resulting after the application of transformations (2) – (4). We will refer to addresses compared in general equations outside an abstracted memory as *control addresses*, and will call those general equations *control equations*. In the cases when the read address in a level of forwarding is a control address, but the write address is not or the write address is also a control address but does not appear in a control equation together with the read address, then that level of forwarding is abstracted with uninterpreted function f_{ud} (where "*ud*" stands for "update data"). UF f_{ud} takes 4 argument terms—write address *wa*, write data *wd*, read address *ra*, and data *rd* read from the previous memory state before the write—such that the functionality abstracted with $f_{ud}(wa, wd, ra, rd)$ is $ITE(ra = wa, wd, rd)$. Finally, when the read address is not a control address, the read is abstracted as before based on transformations (2) – (4).

Note that the initial state of the pipeline latches in the implementation processor consists of a domain variable for every term signal (including register identifiers) and a Boolean variable for every Boolean signal. Hence, the equality comparisons of register identifiers done in the stalling logic during the single cycle of regular symbolic simulation along the implementation side of the commutative correctness diagram will be at the level of domain variables serving as register identifiers. Furthermore, using Burch's controlled flushing [7], it is possible to flush the implementation processor without introducing additional register id equality comparisons due to the stalling logic. In controlled flushing, instructions are artificially stalled by overriding the processor stall signals with user-controlled auxiliary inputs until it is guaranteed that the stall signals will evaluate to **false**, i.e., all the data operand values can be provided correctly by the forwarding logic or can be read directly from a register file. Note that we can modify the processor logic during flushing, as all that logic does then is to complete the partially executed instructions. Mistakes in such modifications can only result in false negatives. The symbolic simulation of the non-pipelined specification processor along the specification side of the commutative correctness diagram does not result in additional control equations, as all data operands are read directly from the register files. Therefore, using controlled flushing and applying transformations (2) – (4) will result in an EUFM correctness formula with only those general (control) equations over reg-

ister identifier terms that are introduced by the stalling logic in the single cycle of regular symbolic simulation in order to preserve the correctness of the memory semantics for a register file. The exact abstraction steps are presented next.

Algorithm for Applying the Hybrid Memory Model:

1. Abstract all memories:
   1.1 use rules (2) – (4) to abstract memories extended with forwarding logic;
   1.2 use UF f_{ud} to abstract levels of forwarding where the initial data is not read from a memory:
   $$ITE(e \wedge (ra = wa), d_1, d_0) \;\rightarrow\; ITE(e, f_{ud}(wa, d_1, ra, d_0), d_0) \tag{5}$$
   where d_0 is neither an application of UF f_r nor an *ITE* expression that has an application of f_r among its leaves;
   1.3 identify the control equations and control addresses.
2. For all applications of UF f_r whose address term is an *ITE* expression, push f_r to the leaves of the address term:
   $$f_r(m, ITE(e, ra_1, ra_2)) \;\rightarrow\; ITE(e, f_r(m, ra_1), f_r(\mathrm{m}, ra_2)) \tag{6}$$
   until every address argument of f_r becomes a domain variable. If an address argument to f_r is an application of an UF, then use the nested *ITE*s scheme to eliminate it and again apply (6) recursively.
3. For those applications $f_r(m, ra)$, where the address term ra is a control address and m is of the form $ITE(e, f_u(m_0, wa, wd), m_0)$, do:
   3.1 if the write address wa is an *ITE* expression, $ITE(c, wa_1, wa_2)$, where some of the leaf terms are control addresses compared for equality to ra in a control equation, then apply the transformation:
   $$\begin{aligned} &f_r(ITE(e, f_u(m_0, ITE(c, wa_1, wa_2), wd), m_0), ra) \;\rightarrow \\ &ITE(c, f_r(ITE(e, f_u(m_0, wa_1, wd), m_0), ra), \\ &\qquad f_r(ITE(e, f_u(m_0, wa_2, wd), m_0), ra)) \end{aligned} \tag{7}$$
   3.2 else, if the write address wa is a control address compared for equality to ra in a control equation, then apply the transformation:
   $$f_r(ITE(e, f_u(m_0, wa, wd), m_0), ra) \;\rightarrow\; ITE(e \wedge (ra = wa), wd, f_r(m_0, ra)) \tag{8}$$
   3.3 else, apply the transformation:
   $$\begin{aligned} &f_r(ITE(e, f_u(m_0, wa, wd), m_0), ra) \;\rightarrow \\ &ITE(e, f_{ud}(wa, wd, ra, f_r(m_0, ra)), f_r(m_0, ra)) \end{aligned} \tag{9}$$
   until every memory argument of f_r becomes a domain variable, i.e., is the initial state of a memory.

Note that Step 3 relies on the assumption that every write to a memory is done under the condition that an enabling formula e is **true**. However, an unconditional write is the case when $e \equiv$ **true**. The soundness of the above transformations is proved as follows.

Proposition 2. *The hybrid memory model, based on uninterpreted functions f_u, f_r and f_{ud}, is a conservative approximation of a memory.*

Proof. If a processor is proved correct with arbitrary uninterpreted functions f_u, f_r and f_{ud}, the processor will be correct for any implementation of these uninterpreted functions, including $f_u(m, wa, wd) \equiv write(m, wa, wd)$, $f_r(m, ra) \equiv read(m, ra)$, and $f_{ud}(wa, wd, ra, rd) \equiv ITE(ra = wa, wd, rd)$ that satisfy the complete memory semantics. ❑

Transformations (2) – (9) separate the effects of the forwarding and stalling logic, modeling conservatively their interaction. The result is a dramatic reduction in the number of e_{ij} Boolean variables required for encoding the equality comparisons of g-term domain variables, as now most of the register identifier terms are used only as inputs to uninterpreted functions, i.e., become p-terms. Indeed, only transformation (8) introduces a general equation over register identifiers. However, exactly the same equation already exists in the EUFM formula outside the abstracted memory, so that the number of general equations over register identifiers is equal to the number of equations generated by the stalling logic in the single cycle of regular symbolic simulation of the implementation processor. Hence, the translation of the EUFM correctness formula to propositional logic by applying transformations (2) – (9) will depend on significantly fewer Boolean variables and its tautology checking will be done much faster, compared to the case when these transformations are not applied.

In our previous work [16][17][19], the two final memory states reached along the two sides of the commutative diagram were checked for equality by generating a new domain variable, performing a read from that address of both final states, and comparing for equality the two resulting data terms. That scheme introduces additional e_{ij} Boolean variables encoding the equality of the final read address with each of the write addresses. The advantage of that comparison method is that it can account for the property of transitivity of equality for the register ids [4][5]. Although that property is not required for the correct benchmarks used in this paper, it is needed in order to avoid false negatives for buggy versions, as well as when verifying out-of-order superscalar processors [20]. In the present paper, the final memory states are compared for equality by applying transformation (3) to the final memory state terms, i.e., abstracting function *write* with UF f_u, and directly comparing for equality the resulting final memory state terms. Indeed, what is verified is that the same sequence of updates is performed under all conditions by both sides of the commutative diagram.

Transformations (2) – (9) are based on the assumption that reads and writes are not reordered and occur in the same sequence along the two sides of the commutative correctness diagram—the case in microprocessors with in-order execution.

4 Example

The transformation rules will be illustrated on the pipelined processor in Fig. 1. It can execute only register-register instructions and has 4 stages: Instruction Fetch (IF), Execute (EX), a Dummy stage (D), and Write-Back (WB). Since there is no forwarding logic to bypass the result of the instruction in D to the instruction in EX, a newly-fetched instruction is stalled if it has a data dependency on the preceding instruction in EX. When set to **true**, signal Flush stops the instruction fetching and inserts bubbles in

the pipeline, thus flushing it by allowing partially executed instructions to complete [6]. The instruction memory, IMem, is read-only and is abstracted with 3 UFs—one for each of the instruction fields source register (SrcReg), op-code (Op), and destination register (DestReg)—and 1 UP—for the valid bit (Valid). UFs ALU and +4 abstract, respectively, the ALU in EX and the PC incrementer in IF. The register file, RegFile, is write-before-read, i.e., the newly-fetched instruction will be able to read the data written by the instruction in WB. The user-visible state elements are PC and RegFile. The processor is compared against a non-pipelined specification (not shown) that consists of the same user-visible state, UFs, and UP. It does not have the 3 pipeline latches, stalling logic, or forwarding logic.

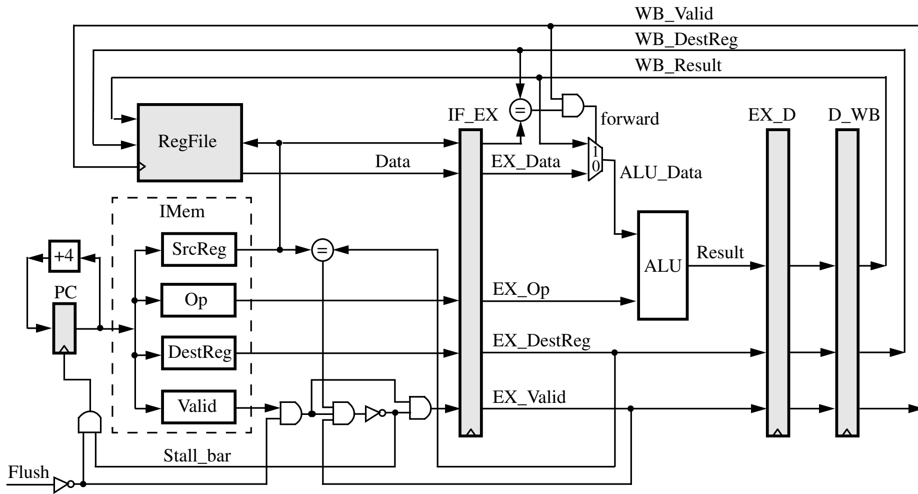

Fig. 1. Block diagram of a 4-stage pipelined processor.

Flushing the pipeline takes 3 clock cycles, as there can be up to 3 instructions in flight. In order to define the specification behavior, the implementation processor is first flushed, reaching the initial specification state $\langle PC_Spec_0, RegFile_Spec_0 \rangle$, as shown below. The specification processor is then exercised for 1 step from that state, reaching specification state $\langle PC_Spec_1, RegFile_Spec_1 \rangle$. *SrcReg*, *DestReg*, and *Op* are new domain variables and *Valid* is a new Boolean variable used to eliminate the single applications of, respectively, the three UFs and one UP that abstract the IMem. Domain variables *PC* and *RegFile* represent the initial state of the corresponding state element. Domain and Boolean variables with prefixes IF_EX_, EX_D_, and D_WB_ represent the initial state of the corresponding pipeline latch. Functions *read* and *write* are already abstracted with UFs f_r and f_u, respectively, according to (2) and (3). The left arrow "←" means assignment.

$$RegFile_0 \leftarrow ITE(D_WB_Valid, f_u(RegFile, D_WB_DestReg, D_WB_Result), RegFile)$$
$$RegFile_1 \leftarrow ITE(EX_D_Valid, f_u(RegFile_0, EX_D_DestReg, EX_D_Result), RegFile_0)$$
$$forward_0 \leftarrow (IF_EX_SrcReg = D_WB_DestReg) \wedge D_WB_Valid$$
$$ALU_Data_0 \leftarrow ITE(forward_0, D_WB_Result, IF_EX_Data)$$
$$Result_0 \leftarrow ALU(IF_EX_Op, ALU_Data_0)$$

$$\begin{array}{ll}
RegFile_Spec_0 & \leftarrow ITE(IF_EX_Valid, f_u(RegFile_1, IF_EX_DestReg, Result_0), RegFile_1) \\
PC_Spec_0 & \leftarrow PC \\
Data_0 & \leftarrow f_r(RegFile_Spec_0, SrcReg) \\
Result_1 & \leftarrow ALU(Op, Data_0) \\
RegFile_Spec_1 & \leftarrow ITE(Valid, f_u(RegFile_Spec_0, DestReg, Result_1), RegFile_Spec_0) \\
PC_Spec_1 & \leftarrow +4(PC)
\end{array}$$

The behavior of the implementation processor is captured by one cycle of regular symbolic simulation (Flush is set to **false**), followed by flushing:

$$\begin{array}{ll}
Stall_bar & \leftarrow \neg(IF_EX_Valid \wedge Valid \wedge (IF_EX_DestReg = SrcReg)) \\
IF_Valid & \leftarrow Valid \wedge Stall_bar \\
Data_1 & \leftarrow f_r(RegFile_0, SrcReg) \\
ALU_Data_1 & \leftarrow ITE(EX_D_Valid \wedge (SrcReg = EX_D_DestReg), EX_D_Result, Data_1) \\
Result_2 & \leftarrow ALU(Op, ALU_Data_1) \\
RegFile_Impl & \leftarrow ITE(IF_Valid, f_u(RegFile_Spec_0, DestReg, Result_2), RegFile_Spec_0) \\
PC_Impl & \leftarrow ITE(Stall_bar, +4(PC), PC)
\end{array}$$

The correctness formula is defined according to (1), given that the processor can fetch up to 1 new instruction and has 2 user-visible state elements—PC and RegFile:

$$\begin{array}{ll}
m_{PC,0} & \leftarrow (PC_Spec_0 = PC_Impl) \\
m_{RegFile,0} & \leftarrow (RegFile_Spec_0 = RegFile_Impl) \\
m_{PC,1} & \leftarrow (PC_Spec_1 = PC_Impl) \\
m_{RegFile,1} & \leftarrow (RegFile_Spec_1 = RegFile_Impl) \\
correctness & \leftarrow m_{PC,0} \wedge m_{RegFile,0} \vee m_{PC,1} \wedge m_{RegFile,1}
\end{array}$$

All of the above expressions are generated by symbolically simulating the implementation and specification processors with a term-level symbolic simulator [21].

Applying rule (4) to expression ALU_Data_1, we get:

$$ALU_Data_1 \leftarrow f_r(RegFile_1, SrcReg)$$

This is achieved by using a unique-expression hash table [16], so that

$$ITE(EX_D_Valid, f_u(RegFile_0, EX_D_DestReg, EX_D_Result), RegFile_0)$$

is identified as the existing identical expression $RegFile_1$.

UF f_{ud} is next applied in order to abstract the one level of forwarding that affects expression ALU_Data_0 (Step 1.2 of the algorithm for the hybrid memory model):

$$\begin{array}{ll}
temp_0 & \leftarrow f_{ud}(D_WB_DestReg, D_WB_Result, IF_EX_SrcReg, IF_EX_Data) \\
ALU_Data_0 & \leftarrow ITE(D_WB_Valid, temp_0, IF_EX_Data)
\end{array}$$

As a result, the only general equation left in the correctness formula is ($IF_EX_DestReg = SrcReg$) in expression $Stall_bar$, where both $IF_EX_DestReg$ and $SrcReg$ are addresses for the abstracted memory RegFile, i.e., they are used as address arguments in applications of f_r and f_u where the initial memory state is domain variable *RegFile*. Hence, that equation is a control equation and domain variables $IF_EX_DestReg$ and $SrcReg$ are control addresses.

Recursively applying Step 3 of the algorithm for the hybrid memory model to expressions $Data_0$ and ALU_Data_1, where the address term $SrcReg$ of f_r is a control address, we get:

$temp_1$	$\leftarrow f_r(RegFile, SrcReg)$
$temp_2$	$\leftarrow f_{ud}(D_WB_DestReg, D_WB_Result, SrcReg, temp_1)$
$temp_3$	$\leftarrow ITE(D_WB_Valid, temp_2, temp_1)$
$temp_4$	$\leftarrow f_{ud}(EX_D_DestReg, EX_D_Result, SrcReg, temp_3)$
$temp_5$	$\leftarrow ITE(EX_D_Valid, temp_4, temp_3)$
$temp_6$	$\leftarrow IF_EX_Valid \wedge (IF_EX_DestReg = SrcReg)$
$Data_0$	$\leftarrow ITE(temp_6, Result_0, temp_5)$
ALU_Data_1	$\leftarrow temp_5$

Now the only general equation left in the correctness formula is (*IF_EX_DestReg* = *SrcReg*). Hence, only these two address terms will be g-terms, while the rest will be p-terms, and just one e_{ij} Boolean variable will be introduced. In contrast, there will be 8 e_{ij} variables if the hybrid memory model is not applied: 1 for encoding the above equation; 3 for the equations between *SrcReg* and each of the destination registers used for writes within $RegFile_Spec_0$ in order to account for the forwarding property when *read*($RegFile_Spec_0$, *SrcReg*) is eliminated (see Sect. 2); and 4 for equations between a new domain variable and each of the 4 destination registers used for writes to the RegFile when its final states are compared for equality (see Sect. 3). The reduction in the number of e_{ij} variables and the speedup become dramatic for complex microprocessors, as shown in Sect. 6.

5 Rewriting Rules for Processors with Multicycle ALUs

Processors with multicycle ALUs require applying an additional transformation on the EUFM correctness formula before the rules from Sect. 3. The problem stems from the uncertain advance, as controlled by signal Done, of the instruction in the Execute stage during the single cycle of regular symbolic simulation—see Fig. 2. Multicycle ALUs are abstracted with a place holder [17][18], where an UF is used to abstract the functionality, and a new Boolean variable is introduced to express the non-deterministic completion of the computation during each cycle. The place holder is forced to complete a new computation on every cycle during flushing. As noted in Sect. 3, modifying the processor during flushing can only result in false negatives.

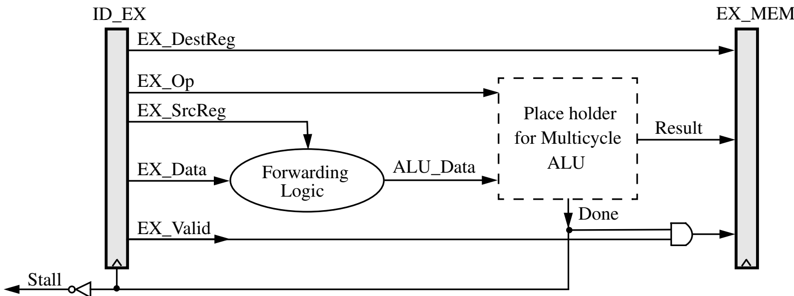

Fig. 2. The Execute stage of a pipelined processor with a multicycle ALU.

Let c be the value of signal Done during the single cycle of regular symbolic simulation. Then, on the next cycle, EX_Data will have the expression *ITE*(c, *read*(m, ra), d_0), where *read*(m, ra) is data that has been read from the register file by the next

instruction, and d_0 is a domain variable for the initial value of EX_Data. Similarly, EX_SrcReg will have the expression $ITE(c, ra, a_0)$, where ra is the source register for the next instruction and a_0 is a domain variable for the initial value of that signal. Assuming one level of forwarding, ALU_Data will have an expression of the kind:

$$ITE(e \wedge (wa = ITE(c, ra, a_0)), d_1, ITE(c, read(m, ra), d_0)) \quad (10)$$

where wa, d_1, and e are, respectively, the destination register, data, and enabling condition for a write in flight to the register file. Since the above expression does not exactly match either of rules (4) and (5), it is rewritten by pulling c to the top of the expression and simplifying the read address along each branch of the new top-level *ITE*:

$$ITE(c, ITE(e \wedge (wa = ra), d_1, read(m, ra)), ITE(e \wedge (wa = a_0), d_1, d_0)) \quad (11)$$

Now the then-expression (selected when c = **true**) of the top-level *ITE* can be rewritten using rule (4), while the else-expression can be rewritten using rule (5).

6 Experimental Results

The benchmarks used for the experiments are the same as in our previous work [17]:

1×DLX-C: A single-issue, 5-stage, pipelined DLX processor [10], capable of executing the 6 instruction types of register-register, register-immediate, load, store, branch, and jump. The 5 stages are Fetch, Decode, Execute, Memory, and Write-Back. Forwarding is used to bypass the data results from the Memory and Write-Back stages to the functional units in the Execute stage. However, forwarding is impossible when a load provides data for the immediately following instruction. In such cases, the data hazard is prevented by a load interlock that stalls the dependent instruction in Decode until the load completes and its result can be forwarded from Write-Back when the dependent instruction is in Execute. There are 2 stalling conditions that can trigger a load interlock—one for each of the two source registers of the instruction in Decode. Stalling due to the second source register is done only when its data value is actually used, e.g., the instruction is not of type register-immediate, so that its immediate data value will be used instead.

2×DLX-CA: A dual-issue, superscalar DLX consisting of two 5-stage pipelines. The first pipeline is complete, i.e., capable of executing the 6 instruction types, while the second pipeline can execute only arithmetic (register-register and register-immediate) instructions. Since load instructions can be executed by the complete pipeline only, there is at most 1 load destination register in the Execute stage, but 4 source registers in the Decode stage for a total of 4 possible load interlock stalling conditions. If a load interlock is triggered due to the first instruction in Decode, both instructions in that stage get stalled, so that the processor fetches 0 new instructions. Else, under a load interlock due to the second instruction in Decode, only the first instruction in that stage is allowed to proceed to Execute, while the second moves to the first slot in Decode and the processor fetches only 1 new instruction to fill the second slot in Decode. Additionally, only the first instruction in Decode is allowed to proceed when its result is used by the second instruction in that stage, or when the second instruction is not arithmetic (i.e., there is a structural hazard), in which case that instruction has to be

executed by the first pipeline. Hence, 0, 1, or 2 new instructions can be fetched each cycle. This design is equivalent to Burch's processor [7].

2×DLX-CC: A dual-issue, superscalar DLX with two complete 5-stage pipelines. Now 2 load destination registers in Execute could provide data for each of the 4 source registers in Decode, for a total of 8 load interlock stalling conditions. When a load interlock is triggered for one of the two instructions in Decode, or when the second instruction in that stage depends on the result of the first, the instructions in Decode proceed as in 2×DLX-CA. However, there is no structural hazard, as both pipelines are complete. Again, 0, 1, or 2 new instructions can be fetched each cycle.

Each of the above processors also has an extension with: branch prediction, marked "**-BP**"; multicycle functional units, "**-MC**," where the Instruction Memory, the ALUs in the Execute stage, and the Data Memory can each take multiple cycles to produce a result; exceptions, "**-EX**," where the Instruction Memory, the ALUs, and the Data Memory can raise an exception; as well as combinations of these features. (For detailed descriptions of these processors and their verification see [17][18].)

The results are presented in Tables 1 and 2. The experiments were performed on a 336 MHz Sun4 with 1.2 GB of memory. The Colorado University BDD package [8], and the sifting BDD variable reordering heuristic [14] were used to evaluate the final propositional formulas. The CPU times are reported for the sequence of symbolic simulation, translation of the EUFM correctness formula to a propositional one, and evaluation of the latter with BDDs. The ratios in the last columns of the tables are of the CPU times before and after applying the automatic abstraction of the register file. Burch's controlled flushing [7] was employed for all of the designs.

The experiments with automatically abstracted register files were run with a breadth-first elimination of the UFs and UPs with the nested *ITE*s scheme (see Sect. 2) when translating the EUFM correctness formula to propositional logic. A variant of the fanin heuristic for BDD variable ordering [12] was used: all nodes in the propositional logic DAG are sorted in descending order of their fanout counts; unless already created, the BDD for each node in that order is built in a depth-first manner, such that the inputs to each AND and OR gate are sorted in descending order of their topological levels and their BDDs are built in that order. The experiments without abstracting the register file were run with various heuristics and optimizations—no single strategy performed uniformly as the best across all benchmarks—and the statistics from the experiment with the minimum CPU time for each benchmark are reported.

As Tables 1 and 2 show, the number of e_{ij} register variables is reduced to at most 10 when automatically abstracting the register file from up to 152 for the most complex benchmark, 2×DLX-CC-MC-EX-BP, before the abstraction. The e_{ij} register variables left after the abstraction are those that encode register id equality comparisons made by the stalling logic only in the single cycle of regular symbolic simulation of the implementation processor. In the case of the single-issue processors, the 2 source registers in Decode are compared with the 1 possible load destination register in Execute, for a total of 2 e_{ij} register variables. 2×DLX-CA and its variants have 4 source registers in Decode and still 1 possible load destination register in Execute (the second pipeline cannot execute load instructions). Furthermore, the 2 source registers of the

Processor	Auto. Abs. Reg. File	BDD Variables					Max. BDD Nodes	Memory [MB]	CPU Time [s]	CPU Time Ratio
		e_{ij}			Other	Total				
		Reg.	Br.	Total						
1×DLX-C		27	0	27	36	63	2,155	5.6	0.26	1.24
	✔	2	0	2	34	36	714	5.4	0.21	
1×DLX-C-BP		27	8	35	41	76	3,408	5.7	0.36	1.24
	✔	2	8	10	39	49	2,224	5.5	0.29	
1×DLX-C-MC		45	0	45	54	99	4,520	5.9	0.75	1.74
	✔	2	0	2	46	48	3,095	5.6	0.43	
1×DLX-C-EX		27	0	27	64	91	7,122	6.4	1.16	1.21
	✔	2	0	2	64	66	5,364	6.0	0.96	
1×DLX-C-MC-EX		36	0	36	77	113	18,108	6.5	4.53	2.35
	✔	2	0	2	76	78	10,313	6.4	1.93	
1×DLX-C-MC-EX-BP		36	10	46	81	127	17,236	6.5	4.06	2.0
	✔	2	10	12	80	92	10,839	6.3	2.03	

Table 1. Statistics for the formal verification of the single-issue pipelined processors. "Auto. Abs. Reg. File" stands for "automatically abstracted register file." The e_{ij} "Reg." variables are the ones that encode equality comparisons between register identifiers. The e_{ij} "Br." variables are the ones that encode equality comparisons between predicted and actual branch address targets.

second instruction in Decode are compared for equality with the destination register of the first instruction in that stage, in order to avoid Read-After-Write hazards [10]. Hence, there are 6 e_{ij} register variables for these benchmarks after abstracting the register file. Processor 2×DLX-CC and its extensions can additionally have a load in the Execute stage of the second pipeline, so that the load interlock logic also compares the destination register of that instruction against the 4 source registers in Decode, for a total of 10 e_{ij} register variables. As expected, most of the register ids have become p-terms after the automatic abstraction of the register file and no longer require Boolean variables for encoding their equality comparisons with other register ids.

The speedup for the single-issue pipelined processors is at most 2.0 times after applying the automatic abstraction of the register file, as these designs are relatively simple and could be verified very efficiently before the abstraction. Indeed, the extra time spent applying the transformation rules on the least complex benchmark, 1×DLX-C, is approximately equal to the time saved in the BDD-based evaluation of the resulting Boolean correctness formula, so that the CPU time was reduced with only 0.05 seconds. However, the speedup becomes dramatic for the dual-issue superscalar benchmarks and ranges from 5.7 to 43.5 times. The maximum number of BDD nodes is also reduced, ranging from 23% for 2×DLX-CA to less than 7% for 2×DLX-CC-MC, relative to the BDD node count without the abstraction.

Benchmark 1×DLX-C was first formally verified by Burch and Dill [6], who required the user to manually provide a case-splitting formula, indicating the conditions under which the processor will fetch and complete 1 new instruction. In order for

Processor	Auto. Abs. Reg. File	BDD Variables					Max. BDD Nodes	Memory [MB]	CPU Time [s]	CPU Time Ratio
		e_{ij}			Other	Total				
		Reg.	Br.	Total						
2×DLX-CA		112	0	112	58	170	17,492	6.8	4.7	5.73
	✔	6	0	6	56	62	4,004	6.2	0.82	
2×DLX-CA-BP		112	18	130	68	198	29,397	7.2	10.5	8.14
	✔	6	22	28	66	94	6,256	6.4	1.29	
2×DLX-CA-MC		142	0	142	76	218	68,895	8.2	21	7.19
	✔	6	0	6	75	81	9,470	6.9	2.92	
2×DLX-CA-EX		122	0	122	92	214	143,330	12	142	21.85
	✔	6	0	6	92	98	21,496	8.6	6.5	
2×DLX-CA-MC-EX		146	0	146	125	271	281,966	14	350	14
	✔	6	0	6	124	130	58,847	9.3	25	
2×DLX-CA-MC-EX-BP		150	61	211	131	342	735,380	22	1,137	18.34
	✔	6	23	29	130	159	96,346	9.3	62	
2×DLX-CC		116	0	116	69	185	40,586	7.7	20	10.2
	✔	10	0	10	65	75	7,823	6.3	1.96	
2×DLX-CC-BP		122	25	147	82	229	74,555	8.8	43	8.21
	✔	10	28	38	78	116	19,988	6.6	5.24	
2×DLX-CC-MC		142	0	142	94	236	315,732	14	226	37.67
	✔	10	0	10	92	102	21,322	7.4	6	
2×DLX-CC-EX		122	0	122	103	225	413,732	18	486	17.36
	✔	10	0	10	102	112	68,159	9.6	28	
2×DLX-CC-MC-EX		146	0	146	150	296	1,229,056	34	3,571	43.55
	✔	10	0	10	148	158	118,216	11	82	
2×DLX-CC-MC-EX-BP		152	35	187	158	345	1,009,206	29	2,593	21.61
	✔	10	32	42	156	198	185,249	12	120	

Table 2. Statistics for the formal verification of the dual-issue superscalar processors. "Auto. Abs. Reg. File" stands for "automatically abstracted register file." The e_{ij} "Reg." variables are the ones that encode equality comparisons between register identifiers. The e_{ij} "Br." variables are the ones that encode equality comparisons between predicted and actual branch address targets.

Hosabettu [11] to formally verify the same benchmark, he needed a month of manual work for the definition of completion functions—one per unfinished instruction, describing how that instruction would be completed, assuming all instructions ahead of it in program order have been completed. The completion functions for all instructions in flight are composed manually in order to compute the abstraction function—mapping an implementation state to a specification state—necessary for the commutative diagram. Ritter, *et al.* [13] could verify the same benchmark after running their symbolic simulator for 65 minutes of CPU time.

Benchmark 2×DLX-CA was first verified by Burch [7] who needed around 30 minutes of CPU time (on a slower Sun4 than the one used for the experiments in this paper) only after manually defining 28 case-splitting formulas and decomposing the commutative correctness diagram into 3 diagrams that are easier to verify. However, that decomposition was subtle enough to warrant the publication of its correctness proof as a separate paper [22]. Hosabettu [11] needed again a month of manual work for the definition of the completion functions for this design. Note that the tool [21] used for the experiments in this paper is completely automatic. It does not require manual intervention except for defining the controlled flushing of the implementation processor—something that takes a couple of minutes and has to be done just once for each design.

7 Conclusions and Future Work

An order of magnitude reduction was achieved in the CPU time for the formal verification of dual-issue superscalar microprocessors with multicycle functional units, exceptions, and branch prediction. That was possible by automatically applying a system of conservative transformation rules for abstracting the register file in a way that separates the forwarding and stalling logic, modeling completely only those levels of forwarding that directly interact with stalling conditions, but abstracting conservatively the rest. The transformation rules are based on the assumption that reads and writes of user-visible state are not reordered and occur in their program sequence.

The effectiveness of a set of conservative rewriting rules depends on accounting for variations in the description style used for the implementation and specification processors, so that false negatives are possible when certain cases are not considered. However, the potential gain from such rewriting rules is a dramatic speedup of up to orders of magnitude, as demonstrated in this paper.

The same transformation rules can be expected to speed up the checking of liveness properties for in-order microprocessors, where a design will be simulated for a fixed number of more than one clock cycles in order to prove that it will eventually complete a new instruction. Future work will extend the transformation rules for application to out-of-order microprocessors.

References

[1] W. Ackermann, *Solvable Cases of the Decision Problem*, North-Holland, Amsterdam, 1954.

[2] R.E. Bryant, “Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams,” ACM Computing Surveys, Vol. 24, No. 3 (September 1992), pp. 293-318.

[3] R.E. Bryant, S. German, and M.N. Velev, “Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic,”[2] ACM Transactions on Computational Logic (TOCL), Vol. 2, No. 1 (January 2001).

[4] R.E. Bryant, and M.N. Velev, “Boolean Satisfiability with Transitivity Constraints,”[2] *Computer-Aided Verification (CAV ’00)*, E.A. Emerson and A.P. Sistla, *eds.*, LNCS 1855, Springer-Verlag, July 2000, pp. 86-98.

[5] R.E. Bryant, and M.N. Velev, “Boolean Satisfiability with Transitivity Constraints,”[2] Tech-

2. Available from: `http://www.ece.cmu.edu/~mvelev`

nical Report CMU-CS-00-101, Carnegie Mellon University, 2000.

[6] 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. http://sprout.stanford.edu/papers.html.

[7] J.R. Burch, "Techniques for Verifying Superscalar Microprocessors," *33rd Design Automation Conference (DAC '96)*, June 1996, pp. 552-557.

[8] CUDD-2.3.0, http://vlsi.colorado.edu/~fabio.

[9] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal, "BDD Based Procedures for a Theory of Equality with Uninterpreted Functions," *Computer-Aided Verification (CAV '98)*, A.J. Hu and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 244-255.

[10] J.L. Hennessy, and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd edition, Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[11] R. Hosabettu, "Systematic Verification of Pipelined Microprocessors," Ph.D. thesis, Department of Computer Science, University of Utah, August 2000. http://www.cs.utah.edu/~hosabett.

[12] S. Malik, A.R. Wang, R.K. Brayton, and A. Sangiovani-Vincentelli, "Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment," *International Conference on Computer-AIded Design (ICCAD '88)*, November 1988, pp. 6-9.

[13] G. Ritter, H. Eveking, and H. Hinrichsen, "Formal Verification of Designs with Complex Control by Symbolic Simulation," *Correct Hardware Design and Verification Methods (CHARME '99)*, L. Pierre and T. Kropf, *eds.*, LNCS 1703, Springer-Verlag, September 1999, pp. 234-249.

[14] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *International Conference on Computer-Aided Design (ICCAD'93)*, November 1993, pp. 42-47.

[15] M.N. Velev, and R.E. Bryant, "Exploiting Positive Equality and Partial Non-Consistency in the Formal Verification of Pipelined Microprocessors,"[2] *36th Design Automation Conference (DAC '99)*, June 1999, pp. 397-401.

[16] M.N. Velev, and R.E. Bryant, "Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic,"[2] *Correct Hardware Design and Verification Methods (CHARME '99)*, L. Pierre and T. Kropf, *eds.*, LNCS 1703, Springer-Verlag, September 1999, pp. 37-53.

[17] M.N. Velev, and R.E. Bryant, "Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction,"[2] *37th Design Automation Conference (DAC '00)*, June 2000, pp. 112-117.

[18] M.N. Velev, and R.E. Bryant, "Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction,"[2] Technical Report CMU-CS-00-116, Carnegie Mellon University, 2000.

[19] M.N. Velev, "Formal Verification of VLIW Microprocessors with Speculative Execution,"[2] *Computer-Aided Verification (CAV '00)*, E.A. Emerson and A.P. Sistla, *eds.*, LNCS 1855, Springer-Verlag, July 2000, pp. 296-311.

[20] M.N. Velev, and R.E. Bryant, "Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors,"[2] *submitted for publication*, 2000.

[21] 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,"[2] *submitted for publication*, 2001.

[22] P.J. Windley, and J.R. Burch, "Mechanically Checking a Lemma Used in an Automatic Verification Tool," *Formal Methods in Computer-Aided Design (FMCAD '96)*, M. Srivas and A. Camilleri, *eds.*, LNCS 1166, Springer-Verlag, November 1996, pp. 362-376.