Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic[1]

Miroslav N. Velev[*]

mvelev@ece.cmu.edu

http://www.ece.cmu.edu/~mvelev

Randal E. Bryant[‡, *]

randy.bryant@cs.cmu.edu

http://www.cs.cmu.edu/~bryant

[*]Department of Electrical and Computer Engineering
[‡]School of Computer Science
Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.

Abstract. We present a collection of ideas that allows the pipeline verification method pioneered by Burch and Dill [5] to scale very efficiently to dual-issue superscalar processors. We achieve a significant speedup in the verification of such processors, compared to the result by Burch [6], while using an entirely automatic tool. Instrumental to our success are exploiting the properties of positive equality [3][4] and the simplification capabilities of BDDs.

1   Introduction

The properties of positive equality [3][4] were proposed as a way to increase the computational efficiency of a decision procedure for the logic of Equality with Uninterpreted Functions and Memories (EUFM). EUFM was introduced by Burch and Dill [5] for verifying of pipelined processors. In collaboration with German [3][4], we recently showed that by extending the syntax of EUFM and by applying certain abstractions, it is possible to use distinct values for all the instruction addresses and data operands. The result is a significantly increased computational efficiency of EUFM.

The main contribution of this paper is in presenting an entirely automatic tool that works on term-level models and is able to handle complex processors, including a dual-issue superscalar DLX [10] with two complete pipelines. We employ a variety of techniques that enhance the performance at each level, including an automatic detection of positive equality comparisons, the encoding method of Goel *et al.* [8] modified to account for positive equality, and an automatic BDD variable ordering. By comparison, in our previous work on positive equality [3][4][20], we only demonstrated the potential of the logic by verifying efficiently single-issue DLX processors implemented at the bit-level. Furthermore, the user was required to define the initial pipeline state and to give hints for the BDD variable ordering.

Our earlier work [18] showed the overhead for verifying bit-level processors with functional units (FUs) implemented at the gate level to be prohibitive for a BDD-based tool, due to the complexity of the generated formulas. The major sources of complexity were the symbolic modeling of all the bits of data in the data path and the feedback loops, created by the forwarding logic. We then employed abstraction and an efficient

---

encoding technique for representing word-level values [19]. While that allowed us to verify more complex designs, we ran into BDD blow-up, due to contradictory BDD variable ordering requirements. Using positive equality and exploiting techniques that make the FUs different for each executed instruction, we succeeded in verifying pipelined processors with very large instructions set architectures [20]. Yet, later we were not successful in scaling these techniques for verifying dual-issue superscalar processors. In this paper we examine abstract term-level models of processors, as has most of the work in this field [5][6][8][12][13]. An area for future research will be to prove that the correctness of an abstract term-level model implies the correctness of the original bit-level design.

The correctness criterion of Burch and Dill's method is presented in Fig. 1. The implementation with transition function F_{Impl} is verified by comparison against a specification with transition function F_{Spec}. On each clock cycle the implementation initiates the execution of between 0 and m instructions, where m is bounded by the issue rate of the processor. In each transition, the specification executes 1 instruction. We use F_{Spec}^{m} to denote m applications of function F_{Spec}. It is assumed that the implementation and the specification start from a pair of matching initial states—Q_{Impl} and Q_{Spec}, respectively—where the match is determined according to an abstraction function Abs. The correctness criterion is that the two transition functions should yield a pair of matching final states—Q'_{Impl} and Q'_{Spec}, respectively—where the match is determined by the same abstraction function. In other words, the abstraction function should make the diagram commute. This correctness criterion is due to Hoare [8] who used it (in a version where $m = 1$) for verifying computations on abstract data types in software.
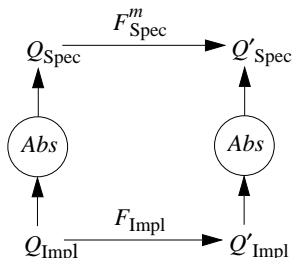
$$Q_{\mathrm{Spec}} \xrightarrow{F_{\mathrm{Spec}}^{m}} Q'_{\mathrm{Spec}}$$
$$\begin{array}{ccc} & Abs & & Abs \\ Q_{\mathrm{Impl}} & \xrightarrow{F_{\mathrm{Impl}}} & Q'_{\mathrm{Impl}} \end{array}$$

Fig. 1. Commutative diagram for the correctness criterion

The correctness criterion, as formulated by Burch [6], is expressed by:

$$\forall Q_{\mathrm{Impl}} \; \exists m \; [Abs(F_{Impl}(Q_{\mathrm{Impl}})) \; = \; F_{\mathrm{Spec}}^{m}(Abs(Q_{\mathrm{Impl}}))]. \tag{1}$$

Burch and Dill's contribution [5] is a conceptually elegant way to automatically compute the abstraction function Abs that maps the pipeline state of a processor to its matching state in the specification by symbolic simulation of the hardware design. Namely, starting from a general symbolic initial state Q_{Impl} they simulate a *flush* of the pipeline by feeding it with bubbles for a sufficient number of cycles to allow all partially executed instructions to complete. Then, they consider the resulting state of the user-visible memories (e.g., the register file and the program counter) to be the match-

ing state Q_{Spec}. Experiments by Isles *et al.* [13] to verify a single-issue pipelined DLX, without using flushing as an abstraction function, ran out of memory, given 1 GB was available.

Burch [6] has extended the method to superscalar processor verification by proposing a flushing mechanism suitable for multi-issue processors and by decomposing the commutative diagram into three commutative diagrams which are easier to verify. A correctness proof of this decomposition is presented in [21]. The weakness of his work is that it requires extensive manual intervention in both decomposing the commutative diagram and in identifying case-splitting expressions, used to speed up the validity checking of the correctness criterion formulas.

Pnueli *et al.* [15] also exploit the positive and negative equality structure in order to reduce the complexity of the decision procedure. Their and our method are not directly comparable, since they do the analysis after eliminating function applications by Ackermann's method [1]. That typically introduces both positive and negative equalities for domain variables, which would only appear as positive equalities in our scheme. Hence, they will not exploit the benefits of positive equality as efficiently as we do by using distinct values for a large number of domain variables.

In the remainder of the paper, Sect. 2 reviews the logic of EUFM. Sect. 3 summarizes the benefits of exploiting positive equality. Sect. 4 presents our algorithm for transforming an EUFM formula into a propositional formula, whose validity implies the validity of the original EUFM formula. Sect. 5 explains our manipulation of the EUFM DAG during the transformation. Experimental results are presented in Sect. 6.

2 Logic of Equality with Uninterpreted Functions and Memories

The logic of Equality with Uninterpreted Functions and Memories (EUFM) presented by Burch and Dill [5] can be expressed by the following syntax:

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

In this logic, *formulas* have truth values while *terms* have values from some arbitrary domain. *Memories* can be viewed as mappings from domain values, representing addresses, to domain or Boolean values (as determined by the type of the memory), representing data. Terms are formed by applications of uninterpreted function sym-

bols, and by applications of *ITE* (for "if-then-else") and *read* operators. The *ITE* operator chooses between two terms based on a Boolean control value, i.e., *ITE*(**true**, x_1, x_2) yields x_1 while *ITE*(**false**, x_1, x_2) yields x_2. The *read* operator takes two arguments, the first of which is a memory, and the second one a term that serves as an address. This operator returns a term for the value of the given memory at the location specified by the address term. A *memory* can be a memory symbol, representing an initial memory state. It can also be the result of an *ITE* operator that selects between two memories. This can be used to express conditional writes to a memory. Finally, a memory can be a *write* operator that takes three arguments, the first of which is a memory, the second is a term that represents a memory address to be updated with the third—a term, representing the new data value for that address. Semantically, *read*(*write*(*memory*, *waddr*, *wdata*), *raddr*) is equivalent to *ITE*(*raddr* = *waddr*, *wdata*, *read*(*memory*, *raddr*)), i.e., a *read* that follows a *write* to a memory returns the value of the *write* when the read and write addresses are equal, and the value of the memory at the read address otherwise. The base case for the *read* operator is to read from the initial state of a memory, represented by a memory symbol *m*, in which case there are no writes to account for, so that the *read* operator can be represented as an uninterpreted function f_m that is specific for memory symbol *m*.

Formulas are formed by comparing two terms for equality, by comparing two memories for equality, by applying the *read* operator to return the contents of the argument memory at the address specified by the argument term, by applying an uninterpreted predicate symbol to a list of terms, and by combining formulas using Boolean connectives. A formula expressing equality between two terms or two memories is called an *equation*. Equations with memory arguments are allowed to occur only in the top-level verification condition to express the equivalence of memory states in the implementation and the specification. The rules for eliminationg reads from memories of type formula are analogous to those for reads from memories of type term, as defined in the previous paragraph, except that reads from initial memory state are represented as uninterpreted predicates.

Every function symbol *f* has an associated *order*, denoted *ord*(*f*), indicating the number of terms it takes as arguments. Function symbols of order zero are referred to as *domain variables*. We use the shortened form *v*, rather than *v*() to denote an instance of a domain variable. Similarly, every predicate *p* has an associated order *ord*(*p*). Predicates of order zero are referred to as *propositional variables*.

The truth of a formula is defined relative to a domain *D* of values and an interpretation *I* of the function, predicate, and memory symbols. An interpretation *I* assigns to each function symbol of order *k* a function from D^k to *D*, to each predicate symbol of order *k* a function from D^k to {**true**, **false**}, and to each memory symbol a function from *D* to *D* or from *D* to {**true**, **false**}, depending on the type of the memory. Given an interpretation *I* of the function and predicate symbols and an expression *E*, we can define the *valuation* of *E* under *I*, denoted *I*[*E*], according to its syntactic structure. *I*[*E*] will be an element of the domain when *E* is a term, and a truth value when *E* is a formula.

A formula *F* is said to be *true under interpretation I* when *I*[*F*] equals **true**. It is

4

said to be *valid over domain D* when it is true for all interpretations over domain *D*. *F* is said to be *universally valid* when it is valid over all domains. It can be shown that if a formula is valid over some suitably large domain, then it is universally valid [1]. In particular, it suffices to have a domain as large as the number of syntactically distinct function application terms occurring in *F*.

3 Positive Equality

In collaboration with German, we have recently shown [3][4] that major improvements can be obtained by exploiting the polarity of the equations in the original formula *F* before replacing any function applications with domain variables. Let us introduce some notation regarding the polarity of equations and their dependent function symbols. For a formula *F* of the form $T_1 = T_2$, we say that this equation is a positive equation of *F*. For formula *F* of the form $\neg F_1$, any positive equation of F_1 is a negative equation of *F*, and any negative equation of F_1 is a positive equation of *F*. For formula *F* of the form $F_1 \wedge F_2$ or $F_1 \vee F_2$, any positive (respectively, negative) equation of either F_1 or F_2 is a positive (respectively, negative) equation of *F* as well. Note that all equations of a formula that controls an *ITE* operator will be both positive and negative equations of a formula containing the *ITE*, since such equations are implicitly negated when selecting the "else-expression" of an *ITE*. We call equations which are both positive and negative in a formula *F*, *general equations* of *F*. Equations of the form $m_1 = m_2$, where m_1 and m_2 are memories, are allowed to occur only as positive equations.

For term *T* of the form $f(T_1, \ldots, T_k)$, function symbol *f* is said to be a data symbol of *T*. For term *T* of the form $ITE(F, T_1, T_2)$, any function symbol that is a data symbol of either T_1 or T_2 is also a data symbol of *T*.

A function symbol *f* is said to be a "p-function" (positive function) symbol of a formula *F* if there are no negative or general equations in *F* for which *f* is a data symbol of one of the equation arguments. All other function symbols are said to be "g-function" (general function) symbols of *F*. Using appropriate abstractions, we can represent all processor operations involving instruction addresses and data operands with p-function symbols, leaving only register identifiers as g-function symbols.

We can exploit the presence of p-function symbols to greatly reduce the number of interpretations that must be considered to determine the universal validity of the original formula. Let Σ denote a subset of the function symbols occurring in *F*. We say that interpretation *I* is diverse with respect to Σ for *F* when for any function application term $f(S_1, \ldots, S_k)$ where $f \in \Sigma$ and any other function application term $g(U_1, \ldots, U_l)$ we have $I[f(S_1, \ldots, S_k)] = I[g(U_1, \ldots, U_l)]$ iff $f = g$ and $I[S_i] = I[U_i]$ for $1 \leq i \leq k$. Interpretation *I* is said to be "maximally diverse" if it is diverse with respect to the set of all p-function symbols in *F*. The following result is from [3][4]:

Theorem 1. *A formula F is universally valid iff it is true in all interpretations that are maximally diverse for F.*

The essential idea behind this theorem is that a maximally diverse interpretation

forms a worst case as far as determining the validity of a formula is concerned. For any less diverse interpretation I, we can systematically derive a maximally diverse interpretation I' such that among the equations, only the positive ones can change their valuations under I', and these can only change from **true** to **false**. Therefore, the valuation of F under the two interpretations must either be equal or satisfy $I[F] = $ **true** and $I'[F] = $ **false**. The proof of the above theorem is presented in [3][4].

4 Transforming an EUFM Formula to a Propositional Formula

We proceed through a series of transformations, starting from the initial EUFM formula, expressing the correctness criterion, and ending with a propositional formula whose validity implies the validity of the original one. At each step we apply various optimizations and simplifications, with the major steps being ordered as:

1. Replace equations of the form $m_1 = m_2$, where m_1 and m_2 are memories, by the equation $read(m_1, a) = read(m_2, a)$, where a is a new domain variable. As defined earlier, such equations can appear only as positive equations in the top-level formula when checking that the two sides of the commutative diagram updated the initial state of a memory in exactly the same way. Since the new domain variable represents an arbitrary address, it is easy to see that if the two sides of the commutative diagram modified that address identically, then they would have modified all addresses identically.

2. Eliminate all *read* operators from updated memory state, as explained in Sect. 2. In our tool we perform this step dynamically as we parse the expressions of the EUFM formula. The result will be that the original read will be replaced by a nested *ITE* expression with a read from the initial state of the memory as a leaf of the expression.

3. Identify the p-function symbols and general function symbols (see Sect. 3).

4. Eliminate UFs and reads from initial memory state (see Sect. 4.1).

5. Translate the reduced EUFM formula to a propositional formula (see Sect. 4.2).

6. Check that the resulting propositional formula is a tautology.

4.1 Elimination of Reads from Initial Memory State and of UFs

Reads from initial memory state and applications of UFs are eliminated in a depth-first way, after all their argument expressions have their reads from initial memory state and UFs eliminated. Specifically, UFs are eliminated by our method of using nested *ITE*s for imposing consistency of the function outputs [19]. Given an UF symbol, say *ALU*, which takes two arguments, with the first eliminated application of this UF being $ALU(T_{11}, T_{12})$, where T_{11} and T_{12} are terms, that UF application is replaced by a new domain variable v_1. Then, the second eliminated application of the same UF, $ALU(T_{21}, T_{22})$, is replaced by $ITE((T_{21} = T_{11}) \wedge (T_{22} = T_{12}),\ v_1, v_2)$, where v_2 is a new domain variable introduced for the case where the new pair of arguments does not equal the previous pair of arguments. Similarly, the third eliminated application of the same UF, $ALU(T_{31}, T_{32})$, is replaced by $ITE((T_{31} = T_{11}) \wedge (T_{32} = T_{12}),\ v_1,\ ITE((T_{31} = T_{21}) \wedge$

$(T_{32} = T_{22})$, v_2, v_3)), where v_3 is a new domain variable introduced for the case where the new pair of arguments does not equal any of the previous pairs of arguments. One can see that the above scheme achieves consistency of the UF's outputs: when $T_{21} = T_{11}$ and $T_{22} = T_{12}$, the second application of the UF *ALU* will evaluate to the value of the first application of *ALU*—the domain variable v_1. The same technique can be used to eliminate applications of an uninterpreted predicate, using new propositional variables instead of domain variables. This transformation is defined formally in [4].

Although a read from initial memory state is semantically equivalent to an uninterpreted function, we handle the translation differently. If the memory is addressed by p-function symbols only, the reads from its initial state are eliminated as applications of an uninterpreted function. However, if a memory is addressed by a g-function symbol, then the reads from its initial state are eliminated by pushing every such read to the leaves of the nested *ITE* address term, i.e., until reaching a domain variable, and introducing a new domain variable for the initial state of the memory at that address. For example, *read(RegFile, ITE(F, reg1, reg2))*, where *reg1* and *reg2* are two domain variables, is transformed to *ITE(F, read(RegFile, reg1), read(RegFile, reg2))* after pushing the read to the leaves of the address term, and *read(RegFile, reg1)* is replaced by the new domain variable *data1*, while *read(RegFile, reg2)* is replaced by the new domain variable *data2*, so that the resulting expression is *ITE(F, data1, data2)*. This can be viewed as initializing the memory for every distinct domain variable that can be selected to be an address term. Note that this technique does not result in equations between two domain variables used as addresses; it is also a conservative approximation since it does not enforce the constraint that the equality of two addresses implies the equality of their initial states. This is one of the keys to the efficiency of our tool. The same scheme can be applied to eliminating uninterpreted function applications as well.

4.2 Translation of the Reduced EUFM Formula to a Propositional Formula

Let F^* be the translation of the original EUFM formula F, resulting after the elimination of *read* and *write* operators, as well as function and predicate applications. Then F^* contains only logic connectives, equations, and *ITE*s, as well as domain and propositional variables.

Our method [3][4] can exploit positive equality by considering only distinct interpretations of the domain variables that are generated when eliminating the p-function symbols. Let Vp be the union of the set of domain variables occurring in F that are p-function symbols, and the set of all new domain variables generated when eliminating the applications of each p-function symbol f. Similarly, let Vg be the union of the set of domain variables occurring in F that are g-function symbols, and the set of all new domain variables generated when eliminating the applications of each g-function symbol h. Let V denote the set of all domain variables in F^*. The following theorem was developed in [3][4]:

Theorem 2. *EUFM formula F is universally valid iff its translation F^* is **true** under all interpretation I^* that are diverse over Vp.*

7

The algorithm that we present next is a modification of the one proposed by Goel *et al.* [8], extended to account for positive equality by considering a variable in Vp to be equal only to itself.

Let $Dep(T)$, the *dependency set of term T*, be the set of domain variables that T may evaluate to. For example, if $T = ITE(b_1, v_1, ITE(b_2, v_2, v_3))$, where v_1, v_2, and v_3 are domain variables, then $Dep(T)$ is $\{v_1, v_2, v_3\}$. For each term T and each variable $v \in Dep(T)$, we generate the formula $E(T, v)$ that represents the conditions under which T would evaluate to v.

For each formula G, we generate a formula $\hat{G}$ which is a propositional translation of G. In the base case for $E(T, v)$, when T is the domain variable v, $E(T, v)$ is **true**. For a term T of the form $ITE(G, T_1, T_2)$, the formula $E(T, v)$ is defined as $\hat{G} \land E(T_1, v) \lor \neg\hat{G} \land E(T_2, v)$. The method of translating G into $\hat{G}$ is as follows:

1. if G is $\neg G_1$ then $\hat{G} \doteq \neg\hat{G}_1$;

2. if G is $G_1 \bullet G_2$ then $\hat{G} \doteq \hat{G}_1 \bullet \hat{G}_2$, where $\bullet$ is either $\land$ or $\lor$;

3. if G is $T_1 = T_2$ then

$$\hat{G} \doteq \bigvee_{v \in Dep(T_1) \cap Dep(T_2)} E(T_1, v) \land E(T_2, v) \quad \lor \quad \bigvee_{\substack{v_i \in Dep(T_1) \cap Vg, \\ v_j \in Dep(T_2) \cap Vg, \\ i \neq j}} E(T_1, v_i) \land E(T_2, v_j) \land e_{min(i,j),max(i,j)}$$

where e_{ij} is a propositional variable introduced to express the equality relation between the g-function domain variables v_i and v_j. Note that we introduce an e_{ij} variable only when v_i and v_j are syntactically distinct variables in Vg. Also, we exploit positive equality by considering variables in Vp to be equal only to themselves—they are used only in the left disjunct of the above formula.

Our propositional formulas do not enforce the transitivity constraints $e_{ik} \land e_{kj} \Rightarrow e_{ij}$, and none of our correct models needed such constraints in verifying them. Note that if a formula F evaluates to **true** without transitivity constraints, it will also evaluate to **true** when such constraints are imposed, e.g., by implication: $(e_{ik} \land e_{kj} \Rightarrow e_{ij}) \Rightarrow F$ where F is already **true**. However, when using BDDs for evaluation of the final propositional formula, we employ the strategy by Goel *et al.* [8] in order to check that a counterexample is not due to a violation of the transitivity constraints. Namely, when the final BDD is not **true**, it is negated in order to express all counterexamples. Given an implicant in the resulting BDD, our tool automatically checks that for each negated variable e_{ij}, there is no sequence $e_{ik_1}, e_{k_1k_2}, ..., e_{k_nj}$ of positive variables that would imply that the negated variable e_{ij} should evaluate to **true**, thus canceling the implicant. The first implicant that is not canceled is printed as a counterexample.

Note that our way of eliminating reads from initial memory state by pushing the reads to the leaves of the address term expressions does not create equations between register identifier domain variables used as address terms in reads from the register file of a processor. This would not be the case if the consistency of the initial memory state

was imposed by Ackermann constraints [1], *read_addr1 = read_addr2* $\Rightarrow$ *init_state1 = init_state2*, as done in [8][15], or by our scheme of using nested *ITE*s [19], where *ITE*(*read_addr1 = read_addr2, init_state1, init_state2*) is returned as the initial state of address *read_addr2* given that *init_state1* was already introduced as the initial state of address *read_addr1*. The result is a reduced number of e_{ij} variables encoding equality relations between domain variables used as register identifiers, which translates into an increased efficiency when evaluating the final propositional formula.

Observe that we are using a conservative approximation by not enforcing consistency of the initial memory state. This makes the verification results sound, but not complete, i.e., false positives would not occur, although false negatives are possible. However, by employing a conservative approximation in our verification, we simplify considerably the propositional formula that has to be checked for being a tautology and, hence, we gain efficiency. We can informally argue that this optimization is complete when verifying our processor models (see Sect. 6) since they do not have direct comparisons of source registers in their control logic. Then, the only way for two source registers to be equal is for them to be simultaneously equal to the same destination register. However, the forwarding logic will then select the result associated with that destination register and would prevent the initial state of the register file from being used. Hence, the consistency of the register file initial state will not matter.

As an implementation note, we can view the set of formulas *E(T, v)* for all $v \in V$, as a very sparse set, i.e., it will simply be **false** for many entries. The usual way to represent such sets is as a list maintained in some canonical order with respect to the domain variables. Then the various operations described above can be implemented by processing these lists to generate either a new list or a single formula.

5 Manipulating the EUFM DAG

When building and transforming the EUFM DAG, we impose several simple structural restrictions in order to achieve maximal sharing of identical expressions. Similar to BDDs [2], we create only one node equal to the constant **true** value and only one node equal to the constant **false** value. We allow only the logic connectives $\land$ and $\lor$, from the possible multi-input connectives. Their inputs are sorted in some canonical order, with duplicates and non-controlling values (**true** for $\land$, and **false** for $\lor$) removed. Expressions of the form $c = a \land b$, where $b = d \land e$, are rewritten as $c = a \land d \land e$, in order to increase the sharing of logically identical expressions. Similar rewritings are done for expressions with the logic connective $\lor$. The presence of a controlling value (**false** for $\land$, and **true** for $\lor$), or the presence of both a and $\neg a$ as inputs, results in returning the controlling value. Otherwise, the list of sorted inputs, together with the type of the connective, forms a key, which is used to search an Operations Hash table for the same expression created previously. If such an expression is not found, it is created and inserted into the Operations Hash table with the formed key.

Other types of expressions—*ITEs*, equations, uninterpreted function applications, and uninterpreted predicate applications, as well as the *read* and *write* operators—also have a key formed in some canonical way in order to access the Operations Hash table.

9

When creating an expression that is the negation of another expression, e.g., $b = \neg a$, where a is not a constant Boolean value, such that the key $\neg a$ is not in the Operations Hash table, we insert two keys in that table: $\neg a$ pointing to expression b, and $\neg b$ pointing to expression a. In this way we ensure that if an expression $c = \neg b$ is created later, it will be identified as expression a. Standard simplifications of *ITE* expressions are also employed, which we omit due to lack of space.

6   Experimental Results

We started with a 5-stage single-issue pipelined DLX [10] model, 1×DLX-C, capable of fetching up to 1 new instruction every clock cycle and implementing the 6 instruction types considered by Burch and Dill [5][6]: register-register, register-immediate, load, store, branch, and jump. The 5 pipeline stages are Fetch, Decode, Execute, Memory, and Write-Back. The pipelined model and its non-pipelined specification were described in our own HDL that uses the primitives of EUFM. Namely, it has support for basic logic gates, multiplexors (*ITE*s), equality comparators, memories, latches, uninterpreted functions, and uninterpreted predicates. The implementation and the specification were simulated with our term-level simulator in order to form an EUFM formula for the correctness criterion. This formula was generated in the SVC script format [17].

The instruction memory of both the implementation and the specification was modeled to produce abstract instructions, consisting of 2 source register identifier terms, 1 destination register identifier term, an immediate datum term, an operation-code term, and 3 Boolean variables used to determine the instruction type. The 3 Boolean variables were decoded by a gate-level PLA to produce the pipeline control signals for the different stages of the pipeline, such that each instruction type gets encoded with a unique binary pattern of the 3 variables (e.g., the register-register instructions are encoded with the pattern 000, the register-immediate with 001, and so on). Therefore, the fetched instructions were restricted to be of only one instruction type, although no assumptions were made about the sequences of executed instructions.

We did not impose any restrictions on the initial state of the pipeline latches, as we did in our previous work with bit-level models [18][20]. Hence, we allow the instruction that is initially in a given pipeline latch to be of all the instruction types simultaneously. Furthermore, we consider initial pipeline states that can never arise in actual operation assuming the pipeline interlocks are correct. By not placing any constraints on the initial state, we cover a larger set of states than is required, but also avoid the need to prove any invariants about the state. Note that if a processor is verified without imposing any restrictions on the instructions in flight, it will also be correct when such restrictions are enforced, e.g., by using the restrictive condition to imply the formula for the correctness criterion, where the formula is already valid. The reason why the processors were verified to be correct without imposing invariants for their initial state is twofold. First, the control logic of our models was not designed to depend on any invariant property of the pipeline state. Second, the pipeline latches that are affected by the interlocks, namely the latches before the Execute, Memory, and Write-Back pipeline stages, get their state reflected on the user-visible memory ele-

ments identically along the two sides of the commutative diagram. Note that these latches cannot be stalled and only transfer their data forward. Hence, the identical initial state of the user-visible memory elements, that the two sides of the commutative diagram start from, is modified in the same way by the state of these three pipeline latches, resulting in new identical state of the user-visible memory elements. Therefore, imposing the invariant properties that hold for a correct pipelined processor was not necessary for the verification of our models.

The operation-code term, produced by the instruction memory for each instruction, was used to identify the instruction sub-type to functional units by being used as an input to functional units, just as some control bits are in actual pipelined processors. Specifically, it was carried through the pipeline stages and used as an input to the ALU in the execution stage (e.g., to discriminate an add from a subtract instruction) and to the uninterpreted predicate determining the condition for a branch to be taken based on the comparison of two data operands (e.g., to discriminate a branch-on-less-than from a branch-on-greater-than instruction). Since the operation-code is not used as an argument to interpreted equality comparators, it gets identified as a p-function symbol by our translation algorithm. Hence, functional units taking the operation-code as an argument get transformed into distinct functional units for each executed instruction after the UF elimination by means of nested *ITE*s. This was observed in our previous work [20], where the same effect was achieved by using the sequential PC (equal to PC + 4) which is also a p-function symbol that uniquely identifies each executed instruction. The result is an increased efficiency of the computation, since the functional consistency of ALUs can be imposed with nested *ITE*s of a few levels of nesting for each executed instruction. Therefore, the overall DAG for the correctness criterion ends up being much simpler, compared to the one where the consistency is maintained by nested *ITE*s for the entire executed instruction sequence.

The data memory was modeled as a Finite State Machine with a latch for storing the present state, as explained in [3]. The result fetched by load instructions was produced by an uninterpreted function, *DMem_Read*, that takes as arguments the present data memory state, the load address, and the operation-code term of the instruction (in this way we modeled byte-level memory accesses). The next data memory state was produced by an uninterpreted function, *DMem_Update*, taking the same three arguments in addition to the data operand which is to be written to the data memory by a store instruction. The next data memory state gets written to the FSM latch under the condition that the instruction is a store instruction that was not squashed by taken branches or jumps. The reason to model the data memory in this way is to prevent the outputs of the ALU from being classified as g-terms, due to their role as addresses of the data memory.

Later, we designed a set of dual-issue superscalar DLX models with in-order execution, having 2 pipelines of 5 stages each:

2×DLX-AA has two arithmetic pipelines (implementing register-register and register-immediate instructions), such that either 1 or 2 new instructions are fetched every clock cycle, conditional on the second instruction in the Decode stage having (or not) a data dependency on the first instruction in that stage;

2×DLX-SA can execute arithmetic and store instructions by the first pipeline and arithmetic instructions by the second pipeline, so that in addition to the case of the above data dependency, 1 instruction will be fetched also when the second instruction in the Decode stage is a store (i.e., there is a structural hazard);

2×DLX-LA can execute arithmetic, store, and load instructions by the first pipeline and arithmetic instructions by the second pipeline, so that 2 load interlocks come into play now (between the instruction in Execute in the first pipeline and the two instructions in Decode) and 0, 1, or 2 new instructions can be fetched each cycle;

2×DLX-CA has a complete first pipeline, capable of executing the 6 instruction types, and an arithmetic second pipeline, such that 0, 1, or 2 new instructions can be fetched each cycle—equivalent to Burch's processor [6];

2×DLX-CS has a complete first pipeline, and a second pipeline that can execute arithmetic and store instructions, such that 0, 1, or 2 new instructions can be fetched each cycle;

2×DLX-CL has a complete first pipeline, and a second pipeline that can execute arithmetic, store, and load instructions, such that 0, 1, or 2 new instructions can be fetched each cycle, conditional on 4 possible load interlocks (between a load in Execute in either pipeline and an instruction in Decode in either pipeline) and the resolution of the structural hazard of branches and jumps in Decode of pipeline two, which need to wait for pipeline one;

2×DLX-CC has two complete pipelines, 4 possible load interlocks, but no structural hazards, such that 0, 1, or 2 new instructions can be fetched each cycle.

Our results are presented in Tables 1, 2, and 3. The experiments were performed on a Sun4 with 10 UltraSPARC-II processors of 336 MHz, having 6 GB of physical memory, and running Solaris 2.6, although we used the computer in a single processor mode. The tautology checking of the final propositional logic formula was done with the Colorado University BDD package [7]. We applied a very simple BDD-variable ordering heuristic. The nodes in the final propositional logic DAG are sorted in decreasing order of their fanout counts, such that if a node is the complement of a Boolean variable, then the fanout count of that node is added to the fanout count of the variable. Note that this merging of fanout counts is done only for Boolean variables. The nodes get their BDDs built according to the sorted order in a depth-first way until either a node with a computed BDD is encountered, or a Boolean variable is reached, which gets declared last in the BDD variable order. Furthermore, the recursive BDD computations for the inputs of an AND (OR) node was discontinued as soon as an input's BDD was evaluated to be 0 (1). In the case of an ITE node, the BDD of the controlling input was computed first, such that if it evaluated to a constant 0 or 1 BDD, only the BDD for the corresponding selected input was computed. Also, we freed BDDs for internal nodes as soon as the BDDs were no longer needed, i.e., as soon as a usage count became equal to the fanout count of the node.

Generating the EUFM formula for the correctness criterion by using our term-level simulator, required less than 1.1 MB of memory and 0.1 seconds of CPU time for

all the processors. We used Burch's controlled flushing [6], where auxiliary inputs are introduced and used only during the flushing of the implementation in order to prevent the pipeline interlocks from introducing uncertainty in the instruction flow during flushing. The controlled flushing significantly reduces the complexity of the expressions for the state of the user-visible memory elements. We found the reduction to be as much as 10 times, in terms of both memory and CPU time, while the effort to add the auxiliary control inputs was negligible, given a familiarity with the designs.

| Processor | DAG Node Counts | | | Topological Levels in Final Propositional Logic DAG |
	Initial EUFM DAG	After eliminating *read*s and UFs	Final propositional logic DAG	
1×DLX-C	299	1,198	334	35
2×DLX-AA	517	2,168	490	21
2×DLX-SA	560	2,534	601	27
2×DLX-LA	623	3,641	1,029	32
2×DLX-CA	759	4,856	1,383	60
2×DLX-CS	779	5,077	1,469	60
2×DLX-CL	796	5,657	1,608	60
2×DLX-CC	850	5,998	1,732	67

Table 1. Statistics from different stages of the translation to a propositional formula. The topological levels in the final propositional logic DAG are computed by assigning a level of 1 to the Boolean variables (the leaves of the DAG). The nodes in the final propositional logic DAG are of types ¬, ∧, ∨, and *ITE*.

| Processor | Final Domain Variables | | Final Vg | | Propositional Variables | |
	$|Vp|$	$|Vg|$	Source Registers	Destination Registers	e_{ij}	Other
1×DLX-C	52	13	7	6	27	36
2×DLX-AA	41	19	9	10	66	16
2×DLX-SA	53	19	9	10	66	20
2×DLX-LA	65	19	9	10	72	25
2×DLX-CA	87	25	13	12	116	46
2×DLX-CS	92	25	13	12	116	48
2×DLX-CL	96	25	13	12	116	51
2×DLX-CC	102	25	13	12	120	57

Table 2. Variable statistics during the translation of the EUFM DAG to a propositional formula. The final p-function domain variable set Vp and the final g-function domain variable set Vg were obtained after eliminating the *read*s and the UFs from the EUFM DAG.

13

Processor	BDD variables	Max. BDD Nodes	Memory [MB]	CPU Time [s]
1×DLX-C	63	2,121	5.8	0.25
2×DLX-AA	82	8,979	6.9	0.46
2×DLX-SA	86	8,319	7.2	0.49
2×DLX-LA	97	11,393	8.3	1
2×DLX-CA	162	163,782	15.4	9
2×DLX-CS	164	188,557	16.3	9
2×DLX-CL	167	236,770	15.9	18
2×DLX-CC	177	433,658	18.2	35

Table 3. Checking the final propositional logic DAG for being a tautology by using BDDs. The BDD variables count is the sum of the counts of e_{ij} and other propositional variables from Table 2.

The best results were obtained after applying an optimization for eliminating common subexpressions in the top-level equations. Given an equation $T_1 = T_2$, where T_1 is a term of the form $ITE(f_{11}, T_{11}, ITE(f_{12}, T_{12}, ... ITE(f_{1k}, T_{1k}, T_3)))$, T_2 is a term of the form $ITE(f_{21}, T_{21}, ITE(f_{22}, T_{22}, ... ITE(f_{2l}, T_{2l}, T_3)))$, and T_3 is a term of nested ITEs—$ITE(f_{31}, T_{31}, ITE(f_{32}, T_{32}, ...))$—that is shared by both T_1 and T_2, then T_3 is replaced by a new domain variable if $Dep(T_3) \notin Dep(T_{1i})$ for $i = 1, ..., k$ and $Dep(T_3) \notin Dep(T_{2j})$ for $j = 1, ..., l$. It can be proved that this optimization is both sound and complete.

As Table 3 shows, our verification times range from less than a second for the single-issue case, up to 35 seconds for the dual-issue superscalar cases. The memory requirement (often the limiting factor for BDD-based applications) ranges from 5.8 to 18.2 MB. The number of propositional variables ranges from 63 to 177, with between 27 and 120 comprising the e_{ij} variables encoding the equality relations between register identifiers. The number of domain variables, identified as p-function domain variables, is between 2 and 4 times greater than that of the g-function domain variables, as illustrated in Table 2.

It should be pointed out that our design 2×DLX-CA is comparable to that used by Burch [6], who could verify his model only after devising 3 different commutative diagrams, providing 28 manual case splits, and using around 30 minutes of CPU time on a SUN4. Therefore, we achieved a speedup of two orders of magnitude. And what is most important of all, we achieve this speedup by an entirely automatic tool.

In order to compare our results to those by Goel *et al.* [8], who proposed the e_{ij} encoding, we ran experiments for verifying the CMU-Pipe, also used by Burch and Dill [5]. CMU-Pipe is a 3-stage pipelined data path, which implements only register-register instructions with 2 source registers and 1 destination register. It has 3 pipeline stages, and 1 level of multiplexors in the forwarding logic. Our tool required less than 1.1 MB of memory and 0.02 seconds of CPU time to verify this benchmark, including the time to simulate it and generate the EUFM formula for the correctness criterion.

We did not use Burch's controlled flushing, which is not applicable since the design does not have interlocks. Furthermore, the correctness criterion formula was evaluated to be valid as soon as our tool was done parsing it, due to our strategy of automatically eliminating reads from updated memory state and using a maximally shared EUFM DAG. The two terms which represent the state of the register file after exercising the implementation and the specification, respectively, simply happen to have exactly the same structure. Using the Operations Hash table helps identify them as exactly the same term, so that when the final equation expression is parsed and its two argument terms are found to be exactly the same expression, the equation is automatically evaluated to be **true**. Hence, BDDs were not used at all. Also, we did not have to exploit positive equality. Goel *et al.* [8] reported CPU time of 0.5 seconds and needed over 130,00 BDD nodes. They used extensive manual intervention in order to impose the constraints for: 1) consistency of the ALU outputs, 2) consistency of the register file initial state, and 3) reflecting a sequence of writes on the initial state of memories. They do not present results from other benchmarks.

Modeling the data memory as a Finite State Machine, as explained earlier in this section, was crucial to the efficiency of our methodology. An alternative way for representing the data memory is to use an uninterpreted function that will serve as a "translation box," accepting the output of the ALU as an input and producing an output that is used to address a regular memory, representing the state of the data memory. In this way, the output of the ALU will still be automatically classified as a p-term, while it will be mapped to a g-term (via the translation box) that will address the memory. Byte-level memory accesses can be modeled by a read-modify-write strategy, by using an uninterpreted function to change the present state of the address. However, when verifying such versions of 2×DLX-CA and 2×DLX-CC, the BDD package ran out of memory after 8 and 5 hours, respectively.

In order to assess the performance of BDDs when verifying incorrect designs, we created 100 versions of 2×DLX-CC, each with a different error. They were all detected by usually using up to twice the CPU time, memory, and BDD nodes required for the verification of the correct processor. However, in the worst case, one of these models did need 1,600 seconds of CPU time, 168 MB of memory, and 8,100,000 BDD nodes.

We also ran experiments using the Stanford Validity Checker (SVC) [17] to evaluate the validity of the EUFM formula for 1×DLX-C. SVC did not finish within 24 hours. Computing the automatically generated correctness criterion (1) using a non-BDD-based validity checker for the logic of EUFM results in a considerable increase in complexity, due to the prohibitive number of case splits that are required even for a simple 5-stage DLX processor. In our BDD-based tool, evaluating the Boolean expression for (1) is made trivial by the simplification capabilities of the BDD package.

Using the SAT-checker GRASP [9][14] as a tautology checker instead of BDDs, resulted in 2 seconds of CPU time for verifying 1×DLX-C, 70 seconds for verifying 2×DLX-AA, and 224 seconds for verifying 2×DLX-SA, and 1:50 hours for verifying 2×DLX-LA. Prover, a commercial SAT/tautology-checker based on Stålmarck's method [16], required 10 seconds of CPU time for verifying 1×DLX-C, 60 seconds for verifying 2×DLX-AA, 5.5 hours for verifying 2×DLX-SA, and more than 24 hours

(the run time limit) for verifying 2×DLX-LA. None of the SAT-checkers was able to verify 2×DLX-CC within 24 hours. Then, we applied these tools to verifying an incorrect version of our last model, 2×DLX-CC. SVC, Prover, and GRASP could not produce a counterexample within 24 hours, while using BDDs for checking the formula of the same incorrect design resulted in generating a counterexample in 37 seconds, consuming 18 MB of memory. Experiments with another SAT-checkers—SATO [22]—showed that it was not more successful than GRASP and Prover. Therefore, BDDs were the most efficient means to verify both correct and erroneous processors.

7 Conclusions

We have achieved considerable speedup in the verification of dual-issue superscalar DLX processors, compared to the result by Burch [6]. Furthermore, our tool is entirely automatic and does not require manual intervention, compared to previous work based on the logic of Equality with Uninterpreted Functions and Memories (EUFM) [5][6][8]. The keys to our success were: 1) exploiting the properties of positive equality [3][4], which allow domain variables used in non-negated equality comparisons to be treated as distinct from any other domain variable; 2) using e_{ij} Boolean variables [8] to represent the outcome of those domain variable equality comparisons, which are used both negated and non-negated in the formula, when translating the EUFM formula to a propositional formula; 3) eliminating the reads from the initial state of memories in a way that does not create equality comparisons between two read addresses; 4) defining the ALUs in the abstract models in a way that will turn them into distinct functional units for each executed instruction, based on the properties of positive equality; 5) manipulating the EUFM DAG in a way that results in a maximal sharing of nodes; and, 6) using BDDs to evaluate the resulting Boolean formula, by applying an efficient BDD variable ordering heuristic.

We also showed BDDs to be unmatched by SVC [17], applied to the original EUFM formula, and by Prover [16], SATO [22], and GRASP [9][14], used as alternative tautology checkers of the propositional logic formulas generated by our tool. In contrast to these four methods based on combinatorial search, BDDs capture the full structure of a problem as a single data structure, rather than repeatedly enumerating and disproving possible counterexamples.

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 Serveys, Vol. 24, No. 3 (September 1992), pp. 293-318.

[3] R.E. Bryant, S. German, and M.N. Velev, "Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions,"[2] *Computer-Aided Verification (CAV'99)*, LNCS, Springer-Verlag, June 1999.

[4] R.E. Bryant, S. German, and M.N. Velev, "Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic,"[2] Technical Report CMU-CS-99-115, Carnegie Mellon University, 1999.

[5] 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. Available from: `http://sprout.stanford.edu/papers.html`.

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

[7] CUDD-2.3.0, URL: `http://vlsi.colorado.edu/~fabio`.

[8] 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.

[9] GRASP, URL: `http://andante.eecs.umich.edu`.

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

[11] C.A.R. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica*, 1972, Vol.1, pp. 271-281.

[12] R. Hojati, A. Kuehlmann, S. German, and R.K. Brayton, "Validity Checking in the Theory of Equality with Uninterpreted Functions Using Finite Instantiations," *International Workshop on Logic Synthesis*, May 1997.

[13] A.J. Isles, R. Hojati, and R.K. Brayton, "Computing Reachable Control States of Systems Modeled with Uninterpreted Functions and Infinite Memory," *Computer-Aided Verification (CAV'98)*, A.J. Hu and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998, pp. 256-267.

[14] J.P. Marques-Silva, and K.A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Transactions on Computers*, Vol. 48, No. 5, May 1999, pp. 506-521.

[15] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel, "Deciding Equality Formulas by Small-Domain Instantiations," *Computer-Aided Verification (CAV'99)*, LNCS, Springer-Verlag, June 1999.

[16] G. Stålmarck, "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), 1989.

[17] Stanford Validity Checker (SVC), URL: `http://sprout.Stanford.EDU/SVC`.

[18] M.N. Velev, and R.E. Bryant, "Verification of Pipelined Microprocessors by Correspondence Checking in Symbolic Ternary Simulation,"[2] *International Conference on Application of Concurrency to System Design (CSD'98)*, IEEE Computer Society, March 1998, pp. 200-212.

[19] M.N. Velev, and R.E. Bryant, "Bit-Level Abstraction in the Verification of Pipelined Microprocessors by Correspondence Checking,"[2] *Formal Methods in Computer-Aided Design (FMCAD'98)*, G. Gopalakrishnan and P. Windley, *eds.*, LNCS 1522, Springer-Verlag, November 1998, pp. 18-35.

[20] 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.

[21] 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.

[22] H. Zhang, "SATO: An Efficient Propositional Prover," *International Conference on Automated Deduction (CADE'97)*, LNAI 1249, Springer-Verlag, 1997, pp. 272-275. Available from: `http://www.cs.uiowa.edu/~hzhang/sato.html`.

---

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

Appendix I: Using the Sifting Variable Reordering Heuristic

After submitting the final version of the paper, we found the sifting dynamic BDD variable reordering heuristic [23], which is implemented in CUDD [7], to result in significant reductions in the maximum number of BDD nodes required for the BDD-based evaluation of the propositional formula. The improved results for three of the benchmarks are shown in Table 4.

Processor	Max. BDD Nodes	Memory [MB]	CPU Time [s]
1×DLX-C	2,127	5.7	0.24
2×DLX-CA	24,227	10.8	6
2×DLX-CC	59,530	14.6	28

Table 4. BDD statistics for three of the benchmarks in Table 3, when the evaluation is done with the sifting dynamic BDD variable reordering heuristic.

While the maximum number of BDD nodes for evaluating the simplest benchmark, 1×DLX-C, is almost unchanged, there is a 7 times reduction in the maximum number of BDD nodes for the other two benchmarks.

We also discovered an error in the simulation sequence for the experiments on modeling the Data Memory with a regular memory and a p-term to g-term translation box for the address terms produced by the ALU. One of the phase clocks was not released, which turned the pipeline latches into transparent latches during one of the simulation clock cycles, and that generated an erroneous EUFM formula for the correctness criterion. After fixing the error, we obtained the results presented in Tables 5, 6, and 7. The extension "-M" of the processor names designates the use of the memory model with a regular memory and a translation box for the address terms. Note that this memory model is a conservative approximation of an actual memory. If a processor is correct with this model, it will also be correct when the uninterpreted function that implements the translation box is replaced with the identity function, i.e., a direct connection that passes the unmodified input term to the output.

Processor	DAG Node Counts			Topological Levels in Final Propositional Logic DAG
	Initial EUFM DAG	After eliminating *read*s and UFs	Final propositional logic DAG	
1×DLX-C-M	346	1,575	468	40
2×DLX-CA-M	855	6,045	1,917	89
2×DLX-CC-M	1,024	8,450	2,698	99

Table 5. Statistics from different stages of the translation to a propositional formula for the experiments on modeling the Data Memory with a regular memory and a translation box.

18

Processor	Final Domain Variables		Final Vg			Propositional Variables	
	$|Vp|$	$|Vg|$	Source Registers	Destination Registers	Other	e_{ij}	Other
1×DLX-C-M	52	20	8	6	6	49	36
2×DLX-CA-M	87	35	14	12	9	165	46
2×DLX-CC-M	102	38	14	12	12	200	57

Table 6. Variable statistics during the translation of the EUFM DAG to a propositional formula for the experiments on modeling the Data Memory with a regular memory and a translation box.

The category "Other" g-terms of the "Final Vg" set in Table 6 consists of the g-terms produced by the translation box in the memory model. The equality comparisons between these terms resulted in more than 50% increase in the number of e_{ij} variables for the most complex benchmark, 2×DLX-CC-M, relative to its version with the FSM model of the Data Memory (see Table 3).

Processor	BDD variables	Without Dynamic BDD Variable Reordering			With the Sifting Heuristic for Dynamic BDD Variable Reordering		
		Max. BDD Nodes	Memory [MB]	CPU Time [s]	Max. BDD Nodes	Memory [MB]	CPU Time [s]
1×DLX-C-M	85	4,637	6	0.35	4,004	5.9	0.52
2×DLX-CA-M	211	4,427,745	92	153	82,008	14	44
2×DLX-CC-M	257	Out of Memory		>5.5h	1,760,815	40	3,007

Table 7. Checking the final propositional logic DAG for being a tautology by using BDDs for the experiments on modeling the Data Memory with a regular memory and a translation box. The experiment with 2×DLX-CC-M without dynamic BDD variable reordering required more than 1.8 GB, the amount of memory and available swap space.

As Table 7 shows, the sifting dynamic BDD variable reordering heuristic resulted again in a considerable improvement of the performance, compared to the case when no dynamic BDD variable reordering is used. However, the maximum number of BDD nodes and the CPU time for the most complex benchmark, 2×DLX-CC-M, are 60 and 100 times bigger, respectively, compared to the results from verifying the version with the FSM model of the Data Memory (see Table 4). Therefore, the significance of the FSM model of a data memory for the efficient verification of our processor models.

References

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

Appendix II: Correctness Criterion for Multi-Issue Processors

We can eliminate the existential quantification in the correctness criterion formula (1) by forming a disjunction over the possible values of *m*. We can also eliminate the universal quantification by making it implicit. Then, a dual-issue processor will have the correctness formula:

$$Abs(F_{Impl}(Q_{Impl})) = Abs(Q_{Impl}) \qquad \vee$$
$$Abs(F_{Impl}(Q_{Impl})) = F_{Spec}(Abs(Q_{Impl})) \qquad \vee$$
$$Abs(F_{Impl}(Q_{Impl})) = F_{Spec}^2(Abs(Q_{Impl})), \tag{2}$$

which is illustrated in Fig. 2, where *equality0*, *equality1*, and *equality2* stand for the fisrt, second, and third disjunct above, respectively, and represent the conditions that the 2-issue processor will execute 0, 1, and 2 instructions, respectively. Given that the implementation has three user-visible state elements—a program counter *PC*, a register file *RegFile*, and a data memory *DMem*, we would define:

$$equality0 \doteq (PC'_{Spec} = PC_{Spec0}) \wedge$$
$$(RegFile'_{Spec} = RegFile_{Spec0}) \wedge (DMem'_{Spec} = DMem_{Spec0}), \tag{3}$$
$$equality1 \doteq (PC'_{Spec} = PC_{Spec1}) \wedge$$
$$(RegFile'_{Spec} = RegFile_{Spec1}) \wedge (DMem'_{Spec} = DMem_{Spec1}), \tag{4}$$
$$equality2 \doteq (PC'_{Spec} = PC_{Spec2}) \wedge$$
$$(RegFile'_{Spec} = RegFile_{Spec2}) \wedge (DMem'_{Spec} = DMem_{Spec2}). \tag{5}$$

The above expressions ensure that all the state elements have changed "in sync." Forming the correctness criterion formula for a processor that can fetch more than 2 instructions per clock cycle is done in a similar way.
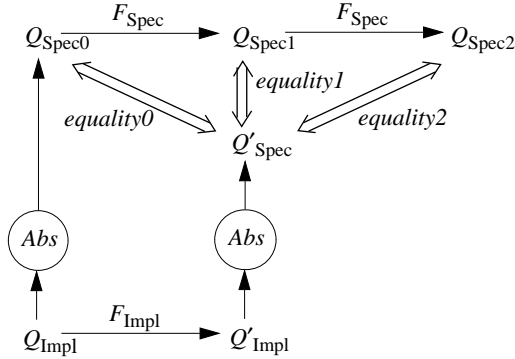

Fig. 2. Commutative diagram for the correctness criterion of dual-issue processors

Note that the correctness criterion for a processor that can execute either 0 or 1 instruction every clock cycle is *equality0* $\vee$ *equality1*. In contrast, when checking the correctness of such a processor, Burch and Dill [5] would require the user to provide a

Boolean formula, *executed1*, indicating the conditions for that processor to have executed 1 instruction. Then, they would check the validity of the two formulas *executed1 $\Rightarrow$ equality1* and *¬executed1 $\Rightarrow$ equality0*. However, forming the correct formula for *executed1* is not easy in a pipelined processor, since the new instruction fetched by the implementation might traverse the pipeline in many ways, due to different interlock conditions, and in each of them might get squashed in a number of pipeline stages, due to taken jumps or branches. Needless to say, forming the formula *executed1* requires the intervention of a highly qualified user. Furthermore, forming manually similar formulas for the number of executed instructions in a multi-issue processor becomes extremely difficult.

Computing the automatically generated correctness criterion (2) using a non-BDD-based validity checker for the logic of EUFM will result in a considerable increase in complexity, due to the prohibitive number of case splits that are required even for a simple 5-stage DLX processor, as was shown in Sect. 6. In our BDD-based tool, evaluating the Boolean expression (2) is made trivial by the simplification capabilities of the BDD package.