Microprocessor Verification Using Efficient Decision Procedures for a Logic of Equality with Uninterpreted Functions[*]

Randal E. Bryant[1], Steven German[2], and Miroslav N. Velev[3]

[1] Computer Science, Carnegie Mellon University, Pittsburgh, PA
Randy.Bryant@cs.cmu.edu
[2] IBM Watson Research Center, Yorktown Hts., NY
german@watson.ibm.com
[3] Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA
mvelev@ece.cmu.edu

Abstract. Modern processors have relatively simple specifications based on their instruction set architectures. Their implementations, however, are very complex, especially with the advent of performance-enhancing techniques such as pipelining, superscalar operation, and speculative execution. Formal techniques to verify that a processor implements its instruction set specification could yield more reliable results at a lower cost than the current simulation-based verification techniques used in industry.

The logic of equality with uninterpreted functions (EUF) provides a means of abstracting the manipulation of data by a processor when verifying the correctness of its control logic. Using a method devised by Burch and Dill [BD94], the correctness of a processor can be inferred by deciding the validity of a formula in EUF describing the comparative effect of running one clock cycle of processor operation to that of executing a small number (based on the processor issue rate) of machine instructions.

This paper describes recent advances in reducing formulas in EUF to propositional logic. We can then use either Binary Decision Diagrams (BDDs) or satisfiability procedures to determine whether this propositional formula is a tautology. We can exploit characteristics of the formulas generated when modeling processors to significantly reduce the number of propositional variables, and consequently the complexity, of the verification task.

1 Introduction

Microprocessors are among the most complex electronic systems created today. High performance processors require millions of transistors and employ exotic techniques

[*] This research was supported at Carnegie Mellon University by SRC Contract 98-DC-068 and by grants from Fujitsu, Motorola, and Intel. This paper will be presented at *Tableaux '99*, June, 1999.

such as pipelining, multiple instruction issue, branch prediction, speculative and/or out-of-order execution, register renaming, and many forms of caching [HP96]. When correctly implemented, these implementation artifacts should be invisible to the user. The processor should produce the same results as if it had executed the machine code in strict, sequential order.

Design errors can often lead to violations of the sequential semantics. For example, an update to a register or memory location by one instruction may not be detected by an instruction following too closely in the pipeline. An instruction following a conditional branch may be executed prematurely, modifying a register even though the processor later determines that the branch is taken. Such *hazard* possibilities increase dramatically as the instruction pipelines increase in both depth and width.

Historically, microprocessor designs have been validated by extensive simulation. Instruction sequences are executed, in simulation, on two different models: a high-level model describing the desired effect of each instruction and a low-level model capturing the detailed pipeline structure. The results from these simulations are then compared for discrepancies. The instruction sequences may be taken from actual programs or synthetically generated to exercise different aspects of the pipeline structure [KN96].

Validation by simulation becomes increasingly costly and unreliable as processors increase in complexity. The number of tests required to cover all possible pipeline interactions becomes overwhelming. Furthermore, simulation test generators suffer from a fundamental limitation due to their use of information about the pipeline structure in determining the possible interactions in an instruction sequence that need to be simulated. A single conceptual design error can yield both an improperly-designed pipeline and a failure to test for a particular instruction combination.

As an alternative to simulation, a number of researchers have investigated using formal verification techniques to prove that a pipelined processor preserves the semantics of the instruction set model. Formal verification has the advantage that it demonstrates correct execution for all possible instruction sequences. Given the large amount of resources currently spent simulating processors, formal verification tools hold the promise of producing more reliable results at a lower cost.

Most of the complexity in modern processors comes from their control logic. The processing of data is localized to a few subsystems such as the arithmetic logic unit and the floating point unit. These can be formally verified separately. We can therefore create an abstract model of the processor that captures the complexities of the control logic while ignoring the details of the data processing. We view program data and addresses as symbolic "terms" having no specified mathematical properties other than the ability to compare two values for equality. We abstract the functionality of data processing blocks as *uninterpreted functions*, with no specified properties other than "functional consistency," i.e., that applications of a function to equal arguments yield equal results: $x = y \Rightarrow f(x) = f(y)$.

Earlier work on formal verification of processors requires detailed analysis of the pipelined structure, e.g., using automated theorem provers [SB90]. Our interest is in developing automated techniques that apply powerful symbolic evaluation techniques to analyze the behavior of the processor over all possible operating conditions. We believe that high degrees of automation are essential to gaining acceptance by chip designers.

Burch and Dill [BD94] were the first to demonstrate that automated decision procedures for a logic of equality with uninterpreted functions (EUF) could be used to verify pipelined processors. They assume there are two abstract models of the processor—a "program" model providing a direct implementation of the instruction set, and a "pipeline" model that captures the complexities of the actual implementation. Verifying that the pipelined processor has behavior matching that of the program model can be performed by constructing a formula in EUF that compares for equality the terms describing the modifications to the programmer-visible state (i.e., the registers, data memory, and program counter) produced by the two models and then proving the validity of this formula.

In their 1994 paper, Burch and Dill also described the implementation of a decision procedure for this logic based on theorem proving search methods. Their procedure builds on ones originally described by Shostak [Sho79] and by Nelson and Oppen [NO80], using combinatorial search coupled with algorithms for maintaining a partitioning of the terms into equivalence classes based on the equalities that hold at a given step of the search. More details of their decision procedure are given in [BDL96].

This paper describes some of our recent results in reducing formulas in EUF to propositional logic in the context of verifying pipelined processors. We show that characteristics of the formulas generated can be exploited to significantly reduce the number of propositional variables and consequently the complexity of proving that the formula is a tautology. By reducing the validity condition to propositional logic, we can apply powerful Boolean methods such as Binary Decision Diagrams (BDDs) [Bry86] as well as highly-optimized satisfiability checkers. By this approach we have achieved much better performance than more classical decision procedures for formulas with uninterpreted functions. More of the technical details are presented in [BGV99b,BGV99a].

2 Verification Methodology

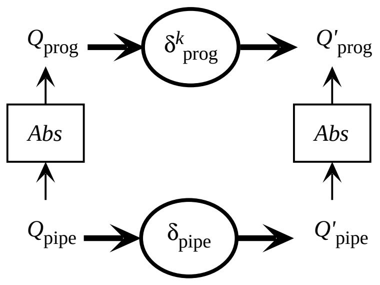
$$\begin{tikzcd}
Q_{\text{prog}} \arrow[r] & \delta^k_{\text{prog}} \arrow[r] & Q'_{\text{prog}} \\
Abs \arrow[u] & & Abs \arrow[u] \\
Q_{\text{pipe}} \arrow[u] \arrow[r] & \delta_{\text{pipe}} \arrow[r] & Q'_{\text{pipe}} \arrow[u]
\end{tikzcd}$$

Fig. 1. Correctness criterion for verifying that pipelined processor "pipe" preserves the sequential semantics of the machine-level language program "prog".

Our task is to verify that a processor will execute all possible instruction sequences properly. Since there is an infinite number of possible sequences, this condition cannot be proved directly. Instead, we show that each possible individual instruction will be executed correctly, regardless of the preceding and following instruction sequences. The correct execution of a complete sequence then follows by induction on its length. One approach to proving the correctness of individual instructions is based on proving the invariance of an abstraction function between processor and program states by each instruction execution. A similar method was proposed by Hoare for proving the correctness of each operation in the implementation of an abstract data type [Hoa72].

We model the processor as having states in the set $\mathcal{Q}_{\rm pipe}$, and the behavior of the processor for each clock cycle of operation by a next-state function $\delta_{\rm pipe}\colon \mathcal{Q}_{\rm pipe} \rightarrow \mathcal{Q}_{\rm pipe}$. Similarly, the state visible to the assembly language programmer (typically the main memory, integer and floating point registers, program counter, and other status registers) is modeled by a state set $\mathcal{Q}_{\rm prog}$ and the execution of a single program instruction by a next-state function $\delta_{\rm prog}\colon \mathcal{Q}_{\rm prog} \rightarrow \mathcal{Q}_{\rm prog}$. In our simplifi ed formulation, we we do not consider the input or output to the processor, but rather that the action taken on each step is determined by the program or pipeline state.

Our task is to show a correspondence between the transformations on the pipeline state by the processor and on the program state by the instruction execution model. This correspondence can be described by an *abstraction function* $Abs\colon \mathcal{Q}_{\rm pipe} \rightarrow \mathcal{Q}_{\rm prog}$ identifying which program state is represented by a given pipeline state. Typically, this corresponds to the effect of completing any instructions in the pipeline without fetching any new instructions. For each pipeline state, there must be a value k indicating the number of program instructions fetched in a given cycle that are ultimately executed. For example, classical RISC pipelines have $k \leq 1$, while superscalar pipelines have k bounded by their "issue rate," typically between 2 and 8. In some pipeline states, we will have a value of k less than its maximum (including possibly $k = 0$). This can occur when instructions must be stalled due to resource conflicts or data dependencies. It also occurs when instructions are fetched and partially executed, but their results are discarded, e.g., due to a mispredicted branch.

The fi rst verifi cation condition [Bur96], is the "correspondence" property illustrated in Figure 1:

$$\forall Q_{\rm pipe} \in \mathcal{Q}_{\rm pipe} \exists k \left[\delta^k_{\rm prog}(Abs(Q_{\rm pipe})) \;=\; Abs(\delta_{\rm pipe}(Q_{\rm pipe}))\right] \qquad (1)$$

where $\delta^k_{\rm prog}$ denotes the k-fold composition of $\delta_{\rm prog}$. Since k is bounded by a small integer, we can eliminate the existential quantifi cation in this equation by forming a disjunction over the possible values of k. For example, a dual-issue pipeline would have the verifi cation condition:

$$\forall Q_{\rm pipe} \in \mathcal{Q}_{\rm pipe} \begin{bmatrix} Abs(Q_{\rm pipe}) = Abs(\delta_{\rm pipe}(Q_{\rm pipe})) & \vee \\ \delta_{\rm prog}(Abs(Q_{\rm pipe})) = Abs(\delta_{\rm pipe}(Q_{\rm pipe})) & \vee \\ \delta_{\rm prog}(\delta_{\rm prog}(Abs(Q_{\rm pipe}))) = Abs(\delta_{\rm pipe}(Q_{\rm pipe})) & \end{bmatrix} \qquad (2)$$

We require as a second verifi cation condition that Abs be surjective to guarantee that all program behaviors can be realized. That is, for every program state $Q_{\rm prog}$, there must be a state $Q_{\rm pipe}$ such that $Abs(Q_{\rm pipe}) = Q_{\rm prog}$.

We require as a third verifi cation condition a "liveness" property that guarantees the processor can always make forward progress. Otherwise we could successfully "verify" a processor that never changes state, giving $k = 0$. This can be expressed by the verifi cation condition:

$$\forall Q_{\text{pipe}} \in \mathcal{Q}_{\text{pipe}} \left[\begin{array}{l} [\delta_{\text{prog}}(Abs(Q_{\text{pipe}})) \neq Abs(Q_{\text{pipe}})] \\ \quad \Rightarrow \\ \exists k [Abs(Q_{\text{pipe}}) \neq Abs(\delta^{k}_{\text{pipe}}(Q_{\text{pipe}}))] \end{array} \right] \tag{3}$$

That is, as long as the corresponding program state is one in which the program makes forward progress (e.g., it is not repeatedly executing an instruction that jumps to itself), the pipeline will make forward progress within k cycles for some value of k. In this paper, as with most of the research on processor verifi cation, we will focus on the correspondence property given by Equation 1.

Observe that the abstraction function can be arbitrary, as long as it satisfi es the three properties listed above. The soundness of the verifi cation is not compromised by an incorrect abstraction function. That is, an invalid abstraction function will not cause the verifi er yield a "false positive" result, declaring a faulty pipeline to be correct. We can let the user provide us with the abstraction function [BF89,NJB97], but this becomes very cumbersome with increased pipeline complexity. Alternatively, we can attempt to derive the abstraction function directly from the pipeline structure [BD94]. Unlike simulation-based test generation, using information about the pipeline structure does not diminish the integrity of the verifi cation.

Burch and Dill [BD94] fi rst proposed using the pipeline description to automatically derive its own abstraction function. They do this by exploiting two properties found in many pipeline designs. First, the programmer-visible state is usually embedded within the overall processor state. That is, there are specifi c register and memory arrays for the program registers, the main memory, and the program counter. Second, the hardware has some mechanism for "flushing" the pipeline, i.e., to complete all instructions in the pipeline without fetching any new ones. For example, this would occur when the instruction cache misses and hence no new instructions could be fetched. A symbolic simulator, which computes the behavior of the circuit over symbolically-represented states, can automatically derive the abstraction function. First, we initialize the circuit to an arbitrary, symbolic state, covering all the states in $\mathcal{Q}_{\text{pipe}}$. We then symbolically simulate the behavior of a processor flush. We then examine the state in the program visible register and memory elements and declare these symbolic values to represent the mapping Abs. Using similar symbolic simulation techniques, we can also compute the effect of the processor on an arbitrary pipeline state δ_{pipe} and the effect of executing an arbitrary program instruction δ_{prog}. Thus, a symbolic simulator can solve the key problems related to verifying pipeline processors.

3 Logic of Equality with Uninterpreted Functions (EUF)

The logic of Equality with Uninterpreted Functions (EUF) presented by Burch and Dill [BD94] can be expressed by the following syntax:

$$\begin{array}{rl} \textit{term} ::= & \textit{ITE}(\textit{formula}, \textit{term}, \textit{term}) \\ & \mid \textit{function-symbol}(\textit{term}, \ldots, \textit{term}) \\ \textit{formula} ::= & \mathbf{true} \mid \mathbf{false} \mid (\textit{term} = \textit{term}) \end{array}$$

$$| \ (\textit{formula} \land \textit{formula}) \ | \ (\textit{formula} \lor \textit{formula}) \ | \ \neg\textit{formula}$$
$$| \ \textit{predicate-symbol}(\textit{term}, \ldots, \textit{term})$$

In this logic, *formulas* have truth values while *terms* have values from some arbitrary domain. Terms are formed by application of uninterpreted function symbols and by applications of the *ITE* (for "if-then-else") operator. The *ITE* operator chooses between two terms based on a Boolean control value, i.e., *ITE*$(\mathbf{true}, x_1, x_2)$ yields x_1 while *ITE*$(\mathbf{false}, x_1, x_2)$ yields x_2. Formulas are formed by comparing two terms for equality, 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 is called an *equation*.

The *ITE* operator distinguishes this logic from other logics of uninterpreted functions, e.g., that used by Shostak [Sho79]. It can be used to model the behavior of "multiplexors" in hardware as well as the effect of a conditional operation in a program. Observe also that this operation has a formula as an argument. We use truth values to represent control values rather than introducing a separate Boolean data type. As a consequence, our logic allows terms to contain formulas, and vice-versa. Although this nesting of operations can be "flattened" into a more conventional form such as conjunctive normal form, this process can cause the formula to grow exponentially. Instead, we prefer to devise decision procedures that can operate directly on our logic.

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 nonempty domain $\mathcal{D}$ of values and an interpretation I of the function and predicate symbols. Interpretation I assigns to each function symbol of order k a function from $\mathcal{D}^k$ to $\mathcal{D}$, and to each predicate symbol of order k a function from $\mathcal{D}^k$ to $\{\mathbf{true}, \mathbf{false}\}$. 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 $\mathbf{true}$. It is said to be *valid over domain* $\mathcal{D}$ when it is true for all interpretations over domain $\mathcal{D}$. F is said to be *universally valid* when it is valid over all domains.

4 Reducing EUF to Propositional Logic

Ackermann has shown [Ack54] that the universal validity of any EUF formula F can be decided by considering only interpretations over a finite domain. In particular, it suffices to have a domain as large as the number of syntactically distinct function application terms occurring in F. Such a domain provides enough distinct values to capture all possible combinations of equalities and inequalities between terms—the only property of terms that our logic considers.

Ackermann also described a technique for eliminating all applications of function and predicate symbols having nonzero order. Each function application is replaced by a domain variable and then constraints are added to enforce functional consistency. For example, if formula F includes terms $f(x_1)$ and $f(x_2)$, we would introduce domain variables fx_1 and fx_2. We would modify F to use these domain variables rather than their respective function application terms, giving formula F'. The verifi cation condition would then be expressed as $[x_1 = x_2 \Rightarrow fx_1 = fx_2] \Rightarrow F'$. Observe how the antecedent enforces functional consistency. By this method, any EUF formula F can be transformed into a formula F^* containing only domain and propositional variables.

In principle we can therefore reduce any EUF formula F having n distinct function application terms to a propositional logic formula by considering as domain the set of all bit vectors of length m, for some value $m \geq \log_2 n$. Each term is then represented as a vector of n formulas, with each domain variable encoded as a vector of m propositional variables. We implemented a variation on this scheme using ordered Binary Decision Diagrams (BDDs) [Bry86] to represent the Boolean functions encoding the terms and formulas symbolically [VB98]. We were able to verify a simple RISC processor implementing only arithmetic instructions. Unfortunately, we found that the BDDs became too complex as we added memory load and store instructions or branch instructions. The interactions between the terms representing successive instructions created circular constraints on the variable ordering that precluded having a good variable ordering. More recent work by Pnueli *et al* [PRSS99] has shown that by examining the detailed structure of the equations in a formula, much tighter bounds can be obtained on the size of the domain associated with each domain variable.

Goel *et al* [GSZAS98] describe an alternate approach to reducing formulas in a logic of equality with uninterpreted functions to propositional logic. They fi rst use Ackermann's method to replace all function applications with domain variables coupled with constraints to impose functional consistency. They then introduce a propositional variable $e_{i,j}$ for each pair of domain variables x_i and x_j in the formula, encoding whether or not the two variables are equal. Based on these variables they generate a propositional formula for each equation encoding the conditions under which the two argument terms will have equal valuations. From this they can generate a propositional formula describing the conditions under which the original formula evaluates to $\mathbf{true}$. This formula must include constraints to enforce the transitivity of equality among the terms. Their BDD-based implementation of this approach was able to verifying only relatively simple pipelines.

5 Positive Equality

We have recently shown 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 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 (resp., negative) equation of F as well. As we consider all of the equations occurring in F, we will also have those that appear as part of the formulas controlling *ITE* operations. We label these to be both positive and negative.

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 a data symbol of T.

A function symbol f is said to be a *p-function* symbol of formula F if there are no negative equations occurring in F for which f is a data symbol of one of the argument terms. Typically these will be symbols that either are not data symbols of any equation or are data symbols only of the top-level verifi cation conditions. For verifying the correspondence property given by Equation 1, we will see that we can represent all operations involving program data and addresses with p-function symbols. The only function symbols that do not qualify as p-function symbols in our application are those representing register identifi ers.

We can exploit the presence of p-function symbols to greatly reduce the number of interpretations that must be considered to determine universal validity. 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 $1\le i\le k$. Interpretation I is said to be "maximally diverse" if it is diverse with respect to the set of all p-function symbols in F.

Theorem 1. *P-formula F is universally valid if and only if it is true in all maximally diverse interpretations.*

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. For any less diverse interpretation I, we can systematically derive a maximally diverse I' such that among the equations, only the positive ones can change their valuations under I', and these can only change from $\mathbf{true}$ to $\mathbf{false}$. Therefore the valuation of F under the two interpretations must either be equal or have $I[F]=\mathbf{true}$ and $I'[F]=\mathbf{false}$.

6 Eliminating Function Applications

We have devised a method of eliminating function application terms from a formula that differs from that of Ackermann [Ack54]. Our method uses a nested *ITE* structure to capture the functional consistency constraints rather than imposing these as antecedents to the formula. Our method has the advantage that it leads to a direct method to exploit positive equality.

We illustrate our technique for replacing function applications by domain variables with a small example. Let F be an EUF formula containing three terms applying function symbol f: $f(x_1)$, $f(x_2)$, and $f(x_3)$, which we identify as terms T_1, T_2, and T_3, respectively. Let vf_1, vf_2, and vf_3 be domain variables that do not occur in F. We generate new terms U_1, U_2, and U_3 as follows:

$$U_1 \doteq vf_1 \tag{4}$$

$$U_2 \doteq \mathit{ITE}(x_2 = x_1,\, vf_1,\, vf_2)$$
$$U_3 \doteq \mathit{ITE}(x_3 = x_1,\, vf_1, \mathit{ITE}(x_3 = x_2,\, vf_2,\, vf_3))$$

We then eliminate the function applications by replacing each instance of T_i in F by U_i for $1 \leq i \leq 3$. Observe that as we consider interpretations with different values for variables vf_1, vf_2, and vf_3, we implicitly cover all values that an interpretation of function symbol f may yield for the three arguments. The nested *ITE* structure shown in Equation 4 enforces functional consistency.

The general method for eliminating function applications follows that of our example formula. For a function symbol f of nonzero order and having n instances, we generate domain variables $vf_1, vf_2, \ldots, vf_n$. Rather than directly replacing function application term T_i with a domain variable, we generate a nested *ITE* structure comparing the arguments of this application to those of each application term T_j for $j < i$. As we consider different interpretations for the newly-generated domain variables, these nested *ITE* structures implicitly cover all possible interpretations of the function application terms while preserving functional consistency. A similar technique can be used to eliminate all instances of a predicate symbol p, using newly-generated propositional variables $ap_1, ap_2, \ldots$. This process is repeated for all function and predicate symbols yielding a formula F^* that contains only domain and propositional variables.

Our method can exploit positive equality by considering only distinct interpretations of the domain variables that are generated when eliminating the p-function symbols. Defi ne Σ_p to be the set of domain variables occurring in F that are p-function symbols, plus the set of all domain variables of the form vf_i generated when eliminating the applications of each p-function symbol f.

Theorem 2. *EUF formula F is universally valid if and only if its translation F^* is true under all interpretations I^* that are diverse over Σ_p.*

This theorem follows by an inductive application of the following argument. Suppose f is in the set of function symbols Σ, that I is diverse over Σ for formula F, and that we replace all instances of f with nested *ITE* structures involving newly-generated domain variables $vf_1, \ldots, vf_n$ to give a formula F'. Then we can construct an interpretation I' for F' that is diverse over $\Sigma - \{f\} \cup \{vf_1, \ldots, vf_n\}$ such that $I'[F'] = I[F]$. Conversely, for any interpretation I' of F', we can extend it to an interpretation I including an interpretation of function symbol f such that $I[F] = I'[F']$.

We can further simplify the task of determining universal validity by choosing particular domains of suffi cient size and assigning fi xed interpretations to the variables in Σ_p. Let Σ_g be the set of variables occurring in F^* that are not in Σ_p. Let $\mathcal{D}_p$ and $\mathcal{D}_g$ be disjoint subsets of domain $\mathcal{D}$ such that $|\mathcal{D}_p| \geq |\Sigma_p|$ and $|\mathcal{D}_g| \geq |\Sigma_g|$. Let α be any 1–1 mapping $\alpha\colon \Sigma_p \rightarrow \mathcal{D}_p$.

Corollary 1. *Formula F is universally valid if and only if its translation F^* is true for every interpretation I^* such that $I^*(v_p) = \alpha(v_p)$ for every variable v_p in Σ_p, and $I^*(v_g)$ is in $\mathcal{D}_g$ for every variable v_g in Σ_g.*

This property follows because any interpretation I^* that is diverse with respect to Σ_p must provide a 1–1 mapping from the variables in Σ_p to domain values. It must therefore be isomorphic to some interpretation where $I^*(v_p) = \alpha(v_p)$ for every $v_p \in \Sigma_p$.

7 Generating a Propositional Formula

We have reduced the problem of deciding the universal validity of an arbitrary formula to one of determining whether a translated formula F^* containing only domain and propositional variables is true under all interpretations that are diverse with respect to some subset Σ_p of the domain variables in F^*. Our method borrows from [GSZAS98] the idea of introducing propositional variables to encode the equalities between domain variables. In our case, however, we only introduce propositional variables for a subset of the domain variable pairs.

For each pair of domain variables, u and v occurring in F^*, we only need to generate a propositional variable $e_{u,v}$ when both u and v are in Σ_g, and there is some equation $T_1 = T_2$ in F^* such that u appears as a data symbol of T_1 while v appears as a data symbol of T_2, or *vice-versa*. This encoding exploits the property that if either u or v is in Σ_p, we can assume they have distinct interpretations. It also exploits the sparse structure of the equations—we need only consider the relation between pairs of variables that appear as data symbols of terms being compared for equality. We can then construct a propositional formula $\hat{F}$ that is a tautology if and only if formula F^*, and consequently our original EUF formula F, is universally valid.

As with [GSZAS98], formula $\hat{F}$ should include constraints of the form $e_{u,v} \wedge e_{v,w} \Rightarrow e_{u,w}$ to consider only interpretations of these variables that satisfy the transitivity of equality. We have found in verifying microprocessor designs that these constraints can often be omitted—hardware designs do not seem to make use of any principles as mathematically deep as transitivity.

8 Modeling Microprocessors in EUF

Our verifier starts with a “term-level” model of both the pipeline and the program version of the processor. That is, we have already abstracted away details of the datapath, replacing functional units with uninterpreted functions. We represent control signals as formulas and multi-bit signals such as operation codes, register identifiers, memory addresses and data as terms. Each instruction is coded as a collection of formulas and terms based on an instruction format having a 3-bit instruction type field, an opcode, two source and one destination register identifiers, and an immediate data value. The task of proving a formal correspondence between such a model and a more detailed register-transfer level model remains a challenging research problem.

To model the register file, we use the memory model described by Burch and Dill [BD94], creating a nested *ITE* structure to encode the effect of a read operation based on the history of writes to the memory. That is, suppose at some point we have performed k write operations with addresses given by terms $A_1, \ldots, A_k$ and data given by terms $D_1, \ldots, D_k$. Then the effect of a read with address given by the term A is given by the term:

$$\mathit{ITE}(A = A_k, D_k, \mathit{ITE}(A = A_{k-1}, D_{k-1}, \cdots \mathit{ITE}(A = A_1, D_1, f_I(A)) \cdots)) \quad (5)$$

where f_I is an uninterpreted function expressing the initial memory state.

By careful design of the term-level model, we are able to treat all symbols representing opcodes, program data, and memory addresses as p-function symbols and hence the domain variables encoding such values are in Σ_p. The symbols representing register identifi ers, on the other hand, do not satisfy the restrictions we impose on p-function symbols. In particular, the pipeline control must compare the register identifi erss of successive instructions to determine when stall or register forwarding conditions arise. The memory model described by Equation 5 involves equations over address terms that control the outcome of *ITE* operations, and hence any data symbols occurring in such terms are not p-function symbols. This causes no problems for the register fi le, since the addresses are register identifi ers. We cannot use such a memory model to represent the main data memory, however, or we would be unable to use p-function symbols to represent instruction and data addresses. Instead, we use a more abstracted memory model in which the effect of a write operation is to cause an arbitrary change of state (represented by an uninterpreted "memory update" function) for the entire memory. Such a model is a conservative abstraction of a true memory, but it suffi ces for modeling processors that perform their memory operations in program order.

9 Experimental Results

We have verifi ed a variety of pipelined processor designs ranging from a single-issue, 5-stage pipeline similar to the DLX processor [HP96] to a variety of superscalar dual-issue pipelines. The most complex of these can handle all instruction types in either side of the pipeline. Our verifi cation times range from less than 1 second for the single-issue case up to 50 seconds for the superscalar cases. The memory requirement (often the limiting factor for BDD-based applications) ranges from 1.5 to 80 Megabytes. The number of propositional variables ranges from 47 to 189, with between 17 and 129 comprising the $e_{u,v}$ variables encoding the relations between register identifi ers.

By contrast, Burch [Bur96] verifi ed a somewhat simpler dual-issue processor only after devising 3 different commutative diagrams, providing 28 manual case splits, and using around 30 minutes of CPU time. We have particularly found that our BDD-based approach can handle the disjunctive verifi cation condition of Equation 2. Methods based on combinatorial search have unacceptably long run times, unless the disjunction is split into separate cases.

We have also experimented with using several different Boolean satisfi ability (SAT) packages to prove that the complement of our generated propositional formula is not satisfi able. We have found these packages perform very well for the single-issue model, and they can often fi nd counterexamples in complex designs containing errors. However they do not complete even after running for many hours when attempting to verify a correct dual-issue design.

10 Conclusions

When verifying pipelined microprocessors using abstracted data paths, we have found that the properties of the EUF formulas to be proved valid can be exploited to greatly simplify the propositional formulas we generate. As a consequence we have been able to verify complex superscalar pipelines with a high degree of automation.

Binary Decision Diagrams provide a powerful mechanism for verifying complex systems. Compared to methods based on combinatorial search, including both decision procedures for EUF as well as SAT solvers for the propositional translation of the verifi cation condition, BDDs capture the full structure of a problem as a single data structure, rather than repeatedly enumerating and disproving possible counterexamples. Our experience has been that BDDs consistently outperform search-based methods when verifying complex designs.

BDDs can only be applied to tasks that are reducible to either propositional logic or to quantifi ed Boolean formulas. An important area of research is to see what other classes of logic can be effi ciently reduced to one of these forms.

References

[Ack54] W. Ackermann, *Solvable Cases of the Decision Problem*, North-Holland, Amsterdam, 1954.

[BDL96] C. Barrett, D. Dill, and J. Levitt, "Validity checking for combinations of theories with equality," *Formal Methods in Computer-Aided Design (FMCAD '96)*, M. Srivas and A. Camilleri, *eds.*, LNCS 1166, Springer-Verlag, November, 1996, pp. 187–201.

[BF89] S. Bose, and A. L. Fisher, "Verifying Pipelined Hardware Using Symbolic Logic Simulation," *International Conference on Computer Design (ICCD '89)*, 1989, pp. 217–221.

[Bry86] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation", *IEEE Transactions on Computers*, Vol. C-35, No. 8 (August, 1986), pp. 677–691.

[BGV99a] R. E. Bryant, S. German, and M. N. Velev, "Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic," Technical report CMU-CS-99-115, Carnegie Mellon University, 1999. Available electronically as: `http://www.cs.cmu.edu/~bryant/pubdir/cmu-cs-99-115.ps`.

[BGV99b] R. E. Bryant, S. German, and M. N. Velev, "Exploiting positive equality in a logic of uninterpreted functions with equality," *Computer-Aided Verification (CAV '99)*, 1999.

[BD94] 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.

[Bur96] J. R. Burch, "Techniques for verifying superscalar microprocessors," *33rd Design Automation Conference (DAC '96)*, June, 1996, pp. 552–557.

[GSZAS98] 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.

[HP96] J. L. Hennessy, and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd edition Morgan-Kaufmann, San Francisco, 1996.

[Hoa72] C. A. R. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica* Vol. 1, 1972, pp. 271–281.

[KN96] M. Kantrowitz, and L. M. Noack, "I'm Done Simulating; Now What? Verification Coverage Analysis and Correctness Checking of the DECchip 21164 Alpha Microprocessor," *33rd Design Automation Conference (DAC '96)*, 1996, pp. 325–330.

[NO80] G. Nelson, and D. C. Oppen, "Fast decision procedures based on the congruence closure," *J. ACM*, Vol. 27, No. 2 (1980), pp. 356–364.

[NJB97] K. L. Nelson, A. Jain, and R. E. Bryant, "Formal Verification of a Superscalar Execution Unit," *34th Design Automation Conference (DAC '97)*, June, 1997.

[PRSS99] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel, "Deciding equality formulas by small-domain instantiations," *Computer-Aided Verification (CAV '99)*, 1999.

[Sho79] R. E. Shostak, "A practical decision procedure for arithmetic with function symbols," *J. ACM*, Vol. 26, No. 2 (1979), pp. 351–360.

[SB90] M. Srivas and M. Bickford, "Formal Verification of a Pipelined Microprocessor," *IEEE Software*, Vol. 7, No. 5 (Sept., 1990), pp. 52–64.

[VB98] M. N. Velev, and R. E. Bryant, "Bit-level abstraction in the verification of pipelined microprocessors by correspondence checking." *Formal Methods in Computer-Aided Design (FMCAD '98)*, G. Gopalakrishnan and P. Windley, *eds.*, LNCS 1522, Springer-Verlag, November, 1998, pp. 18–35.