Techniques for Verifying Superscalar Microprocessors

Jerry R. Burch
Cadence Berkeley Laboratories

Abstract

Burch and Dill [3] described an automatic method for verifying a pipelined processor against its instruction set architecture (ISA). We describe three techniques for improving this method. We show how the combination of these techniques allows for the automatic verification of the control logic of a pipelined, superscalar implementation of a subset of the DLX architecture.

1 Introduction

Burch and Dill's [3] method for verifying microprocessor control circuitry is based on a logic (known as the quantifier-free subset of first-order logic with equality and uninterpreted functions) that allows for the abstraction of datapath values and operations. That is an advantage over propositional logic, which requires that individual bits be modeled explicitly.

Jones, Dill and Burch [10] extended this work by developing a faster validity checking algorithm. The resulting method is more powerful (when applied to processor verification) than other automatic methods [1, 5] and is more automatic than methods based on theorem provers [4, 6, 9, 12, 13].

This paper has four major contributions. The first is a method for splitting up the verification into parts, as described in section 3. This split makes the logical formulas that need to be checked simpler than they would otherwise be. It is also easier for the user to reason about how to keep the implementation and the specification "in sync" during verification.

The second contribution is a way of constructing abstraction functions that are smaller and easier to check (abstraction functions are described in section 2). This involves making a small modification to implementation by adding extra control inputs so that flushing the processor can be controlled or scheduled (see section 4). The abstraction function is made simpler because any variability of instruction flow during flushing (due to interlocks, etc.) is removed.

The third contribution is a new simplification algorithm that makes it easier to check the validity of the formulas constructed during verification (see section 5). This simplification algorithm combines ideas from logical rewrite rules and from the use of observability don't cares in logic synthesis. When combined with the above two techniques, it can simplify most of the relevant formulas to be identically true. Also, it has complexity polynomial in the size of the input DAG of the expression being simplified. In practice, the CPU time needed grows roughly linearly in the DAG size.

$$
\begin{array}{rcl}
\langle \textit{formula} \rangle & ::= & \mathit{ite}(\langle \textit{formula} \rangle,\ \langle \textit{formula} \rangle,\ \langle \textit{formula} \rangle) \\
& & \mid \neg \langle \textit{formula} \rangle \\
& & \mid (\langle \textit{formula} \rangle \vee \cdots \vee \langle \textit{formula} \rangle) \\
& & \mid (\langle \textit{term} \rangle = \langle \textit{term} \rangle) \\
& & \mid \langle \textit{predicate symbol} \rangle(\langle \textit{term} \rangle, \ldots, \langle \textit{term} \rangle) \\
& & \mid \langle \textit{propositional variable} \rangle \mid \mathit{true} \mid \mathit{false} \\[1em]
\langle \textit{term} \rangle & ::= & \mathit{ite}(\langle \textit{formula} \rangle,\ \langle \textit{term} \rangle,\ \langle \textit{term} \rangle) \\
& & \mid \mathit{write}(\langle \textit{term} \rangle,\ \langle \textit{term} \rangle,\ \langle \textit{term} \rangle) \\
& & \mid \mathit{read}(\langle \textit{term} \rangle,\ \langle \textit{term} \rangle) \\
& & \mid \langle \textit{function symbol} \rangle(\langle \textit{term} \rangle, \ldots, \langle \textit{term} \rangle) \\
& & \mid \langle \textit{term variable} \rangle.
\end{array}
$$

Figure 1: Abstract syntax of our subset of first-order logic.

The final contribution is an empirical result showing how the combination of the above techniques allows for the automatic verification of the control logic of a pipelined, superscalar implementation of a subset of the DLX architecture (see section 6). To our knowledge, this is the first time a superscalar processor model has been formally verified against its instruction set architecture, using either manual or automatic methods.

2 Preliminaries

This section gives a brief overview of the method of Dill *et al.* for verifying processors [3, 10].

Quantifier-free, first-order logic with equality and uninterpreted functions is more expressive than propositional logic but less expressive than full first-order logic. An example of a formula in the logic is:

$$\mathit{ite}(f(a) \neq f(b),\ (a \neq b),\ \mathit{true}).$$

The operator ite stands for "if-then-else". The symbol f is an *uninterpreted function* because we do not have in mind a particular meaning for f. The above formula is *valid*, that is, it is true for every possible assignment of a function to f and values for a and b.

The abstract syntax of the logic is given in figure 1. The ite operator may be used to construct a formula (returning a Boolean value) or a term. The ite operator together with the truth constants true and false is sufficient for representing all Boolean operators. However, logical negation and disjunction are included because they allow for more efficient simplification and validity checking algorithms on the logic.

It is helpful when verifying processors to be able to reason about *stores* (memories) such as register files, caches, or main memory. A store is modeled as a function from an address to the data stored at that address. The operations read and write are used to manipulate stores. The expression $read(s, a)$ is the value of store s at address a. The expression $write(s, a, d)$ is the store that has the value d at address a, and the same value as s for any other address.

This model of stores is very abstract. A store contains no information about the sizes of its addresses or values. If a design can be proved correct under this model, then it is correct for any actual implementation with a known memory size.

When expressions in the logic are manipulated by the verification tool, they are all represented with a single, multi-rooted expression DAG. Common subexpressions are included only once in the DAG, so pointer comparison is sufficient to check whether two expressions are syntactically identical. Typically the expression DAGs are several orders of magnitude smaller than the corresponding expression trees. Rewrite rules are used to simplify expressions, but the expressions are not put into a canonical form. For this verification method, letting the structure of the system be reflected in the structure of the expressions that are constructed is more beneficial than using a canonical form for expressions.

In Burch and Dill's method, the user provides behavioral descriptions of the implementation and specification. For processor verification, the specification (the instruction set architecture) describes how the programmer-visible parts of the processor state are updated when one instruction is executed every cycle. The implementation description should be at the highest level of abstraction that still exposes relevant design issues, such as pipelining.

Each description is automatically compiled into a transition function, which is encoded as a vector of symbolic expressions with one entry for each state variable. Any HDL could be used for the descriptions, given an appropriate compiler. Our prototype verifier used a simple HDL based on a small subset of Common LISP. The compiler translates behavioral descriptions into transition functions through a kind of symbolic simulation.

We write F_{Impl} to denote the transition function of the implementation, and we write F_{Spec} to denote the transition function of the specification. By using appropriate control inputs, almost all pipelined processors can be made to continue execution of instructions already in the pipeline while not fetching any new instructions. This is typically referred to as *stalling* the processor. We define F_{Stall} to be a function from implementation states to implementation states that represents the effect of stalling the implementation for one cycle.

All instructions currently in the pipeline can be completed by stalling for a sufficient number of cycles. This operation is called *flushing* the pipeline. We use the function F_{Flush} to model the act of flushing the pipeline. For some sufficiently large integer k,

$$F_{\text{Flush}} = F_{\text{Stall}}^{k},$$

where F_{Stall}^{k} denotes k applications of the function F_{Stall}. It is possible that an incorrectly designed processor may not flush no matter how many cycles it is stalled. It can be shown that such a processor will not satisfy the correctness criteria given below, however, so the bug would be caught.

Intuitively, the verifier should prove that if the implementation and specification start in any matching pair of states, then the result of executing any instruction will lead to a matching pair of states. The primary difficulty with matching the implementation and specification is the presence of partially executed instructions in the pipeline. Various parts of the implementation state are updated at different stages of the execution of an instruction, so it is not necessarily possible to find a point where the implementation state and the specification state can be compared easily.

This problem is solved by defining an *abstraction function* h that maps an implementation state to a specification state. The function h is given by

$$h(Q_{\text{Impl}}) = proj(F_{\text{Flush}}(Q_{\text{Impl}})),$$

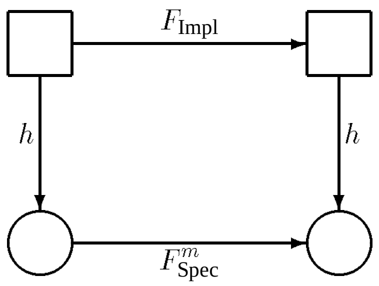

Figure 2: Commutative diagram for the correctness criteria. Squares designate implementation states; circles designate specification states.

where the function $proj$ converts an implementation state to a specification state by simply stripping off all but the programmer-visible parts of the implementation state.[1]

Using the abstraction function h, the correctness criteria is given by

$$\forall Q_{\text{Impl}}\ \exists m\ [h(F_{\text{Impl}}(Q_{\text{Impl}})) = F_{\text{Spec}}^{m}(h(Q_{\text{Impl}}))]. \qquad (1)$$

The integer m is needed to keep the implementation and the specification "in sync". For example, if the processor does not fetch an instruction in state Q_{Impl} (due to a load interlock, say), then m would be zero. For a superscalar processor, m can be greater than one. The correctness condition is represented diagrammatically in figure 2. The diagram in figure 2 is said to *commute* if and only proposition 1 holds.

Recall that each of the functions in proposition 1 is represented by a vector of expressions in quantifier-free first-order logic. The existential quantification is handled by requiring the user to define a *synchronization function* that maps Q_{Impl} to an appropriate value for m. Thus, checking the correctness criteria is reduced to checking the equality of two vectors of quantifier-free expressions (the universal quantification over Q_{Impl} can be made implicit), where the vectors have one entry per specification state variable. Burch and Dill [3] described a validity checking algorithm that can be used to check this equality. Jones, Dill and Burch [10] described ways of significantly speeding up the validity checking algorithm.

In some cases, it may be necessary during verification to restrict the set of implementation states quantified over in proposition 1. In this case, an *invariant* could be provided. It must be checked that the invariant is closed under the implementation transition function, which can be done automatically.

3 Splitting the Commutative Diagram

Our method for splitting the verification task into parts requires that the user provide an additional function, called F_{Sneak}, that maps implementation states to implementation states. The function is intended to model the effect of a *sneak fetch*: the fetching of one instruction into the processor without having any previously fetched instructions flow down the pipeline. In our experience, the only implementation state variables changed by a sneak fetch are the instruction queue and the fetch PC, so F_{Sneak} is much simpler than F_{Impl}. If their is no room for a newly fetched instruction in the current state, then F_{Sneak} does not change the state. If there is an error in the description of F_{Sneak}, then a false negative verification result can occur but not a false positive.

Instead of directly checking the commutative diagram in figure 2, we check the three diagrams in figure 3. We also check

[1]As a technical matter, we require that h be surjective. Automatically checking this is not computationally demanding; the necessary CPU time is included in our empirical results.

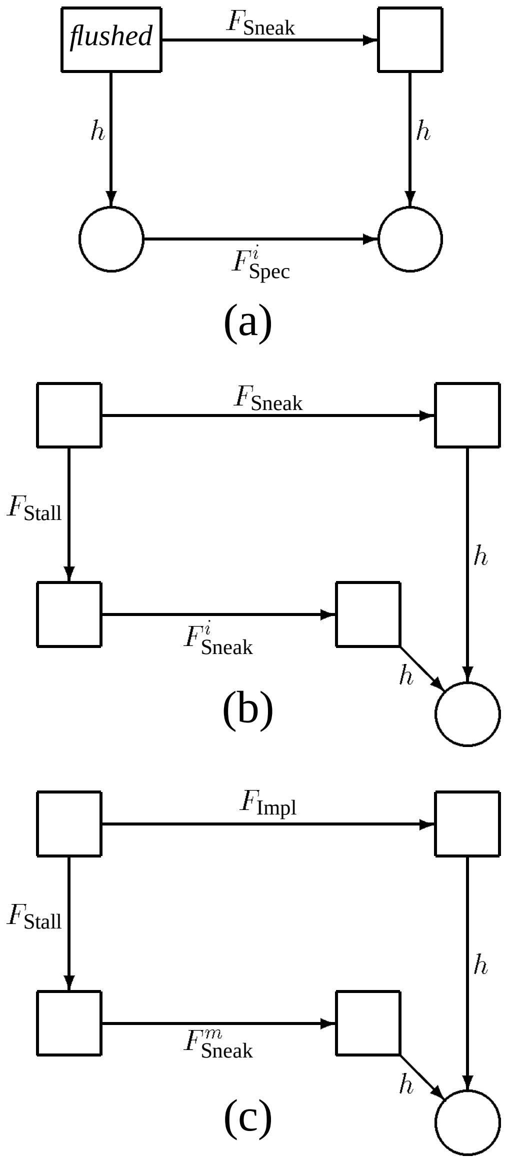

Figure 3: Checking that these diagrams commute is a computationally easier way to check the commutativity of figure 2. The variable i ranges over $\{0, 1\}$ and m ranges from zero to the maximum number of instructions the implementation can fetch in one cycle.

that

$$\forall Q_{\text{Impl}}\ [h(F_{\text{Stall}}(Q_{\text{Impl}})) = h(Q_{\text{Impl}})].$$

We assume that the user has given a predicate that defines when the pipeline is in a flushed state, and has used the validity checker to verify that F_{Flush} always returns a flushed state. Due to space limitations, we do not provide a proof that this check implies that figure 2 commutes.

Computationally, the check of the diagram in figure 3(a) is easy because the initial state is flushed and because the fetching and execution of at most one instruction is involved. The diagrams of figure 3(b) and figure 3(c) are more difficult to check. Even so, they can be much easier to check than figure 2, for the following reason.

Typically, the two implementation states on the right of side of figure 3(c) have the same values for most of their state variables. Thus, after the abstraction function h is applied, the expressions that then need to be checked for equality have similar structure. This structure can be exploited by the validity checker. Similar intuition applies to figure 3(b). However, in that diagram many of state variables of the lower right implementation state differ from the upper right implementation state because of the application of F_{Stall}: the stall cycle causes instructions to have flowed down the pipeline more in one state than in the other. But since the abstraction function h is constructed using F_{Stall}, this difference is largely removed after h is applied. So, as in figure 3(c), the expressions that are checked for equality have similar structure.

4 Constructing Abstraction Functions

Burch and Dill [3] described how to construct an abstraction function automatically by flushing the processor. This section describes a refinement of that method that constructs much simpler abstraction functions while requiring only a little more human intervention.

Figure 4 shows how a standard 5-stage pipeline is flushed as described by Dill *et al.* [3, 10]. In step (b), whether there is in an instruction or bubble in the id or ex stages depends on whether there was a load interlock in step (a). This variability of whether or not an instruction is present is passed down the pipe and ultimately appears in the abstraction function that is constructed. While the extra complexity that results is not great in this case, it is much worse in a superscalar processor, such as the one described in section 6.

The first step in reducing the complexity of the abstraction function is to add an extra control input (called $force\_stall$) to the implementation model. When $force\_stall$ is negated, the processor behaves just as before. However, when $force\_stall$ is asserted, the instruction in the id stage is not allowed to be issued to the ex stage, regardless of whether a load interlock would have occured.

By choosing when $force\_stall$ is asserted, the flushing of the pipeline can be scheduled so that there is no variability in how instructions flow through the pipeline (see figure 5). This results in simpler expressions for the abstraction function, making it easier to check the commutative diagrams in figures 2 or 3.

Since we modified the processor by adding the $force\_stall$ input, there might be some concern that this could lead to erroneous verification results if a mistake was made. Notice, however, that the modified version of the processor is only used when constructing the abstraction function. Thus, a mistake can only lead to a false negative result, not a false positive.

There is a non-trivial amount of human intervention involved in modifying the processor, and devising a flushing schedule. However, the extra effort is more than justified by the increased complexity in the processor models that can be verified. There are a couple of simple rules to follow to make it easier to use this technique. At any stage where an instruction can be stalled in the pipeline, add a control input that can be used to force the stalling of that instruction. Then, schedule the flushing so that one of these control inputs is negated only if the corresponding instruction is guaranteed not to stall.

5 Simplification Algorithm

Applying a rewrite rule to an expression e involves finding a subexpression of e that matches the left hand side of the rewrite rule, and then replacing that subexpression with the right hand side of the rule. Thus, there is a certain locality to applying rewrite rule: only a small part of e is involved in the rewrite. This locality can limit the power of rewrite rules.

We have developed a method for simplifying expressions that avoids this locality, and thereby overcomes some of the limitations of rewrite rules. It is related to logic synthesis methods that use *observability don't cares* [7, 11]. Consider the expression $ite(p, x, y)$.

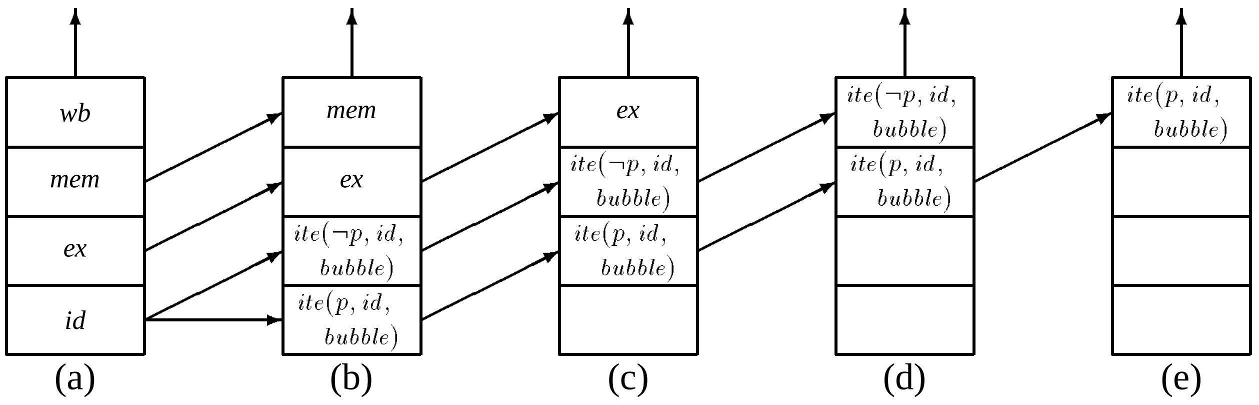

Figure 4: Flushing a pipeline in the straightforward manner.

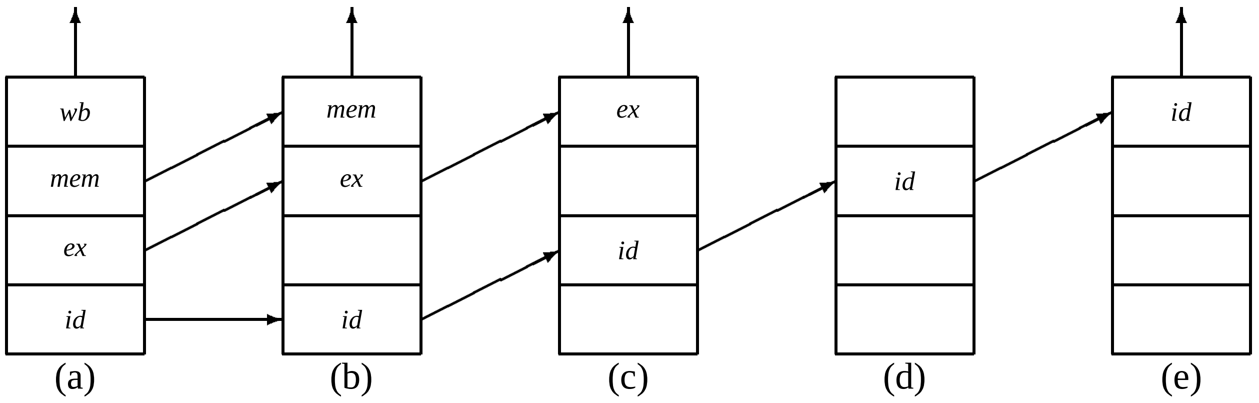

Figure 5: Scheduled pipeline flushing. In step (a), an added control input is used to force a load interlock. In the other steps, the control input is negated and an interlock cannot occur because either the *id* stage or the *exe* stage contains a bubble. As a result, there is no variability in the flow of instructions and the abstraction function that is constructed is simpler than it would otherwise be.

Because the subexpression y cannot be *observed* when p is true, p is called an *observability don't care* of y. We would like to be able to simplify y using the don't care set p. It is easy to produce rewrite rules that can simplify the top level of y, for example:

$$ite(p, x, ite(p, y_1, y_0)) \quad \longrightarrow \quad ite(p, x, y_0).$$

However, we want to be able to use p to efficiently simplify subexpressions deep inside y, regardless of how large y is.

The algorithm in figure 6 describes how this is done. The procedure *simplify* does the actual simplification. It takes an argument u, which is the expression to be simplified, and a care set b, which can be used when simplifying. When *simplify* is used during processor verification, b is typically the universal care set (that is, the don't care set is empty). In that case, the simplification of u comes completely from using local care sets that are constructed for each subexpression of u. The procedures *update_queue* and *build_result* are auxiliary procedures used by *simplify* (see figure 7). For clarity, the algorithm is a simplified version of the one used for processor verification. Instead of handling arbitrary expressions in the logic, it only handles boolean expressions that are constructed using $true$, $false$, boolean variables and the ite operator.

In the algorithm, variables of type *care_set* can be represented with expressions or by some other symbolic method. To understand the algorithm, it is easiest to think of care sets as expressions. The functions *disjoin* and *conjoin* perform the appropriate operations on care sets. These operations can be approximate; the algorithm is sound as long a care set is never underestimated, but care sets can be overestimated. The procedure *provably_empty* returns true if the expression representing a care set can be shown to be tautologically false. This can also be approximated; it is okay for *provably_empty* to return $false$ even if its argument is an empty care set. The procedure call $build\_ite\_expr(p, x, y)$ constructs the expression $ite(p, x, y)$ and then applies basic rewrite rules (such as constant folding, etc.) before returning the expression.

The heart of the algorithm is the while-loop in *simplify*. A subexpression v is pulled off the queue in biggest-first order, which is analogous to using reverse topological order when processing an acyclic circuit netlist. If v is of the form $ite(p, v_1, v_0)$, then it is determined whether the care set c of v allows v to be replaced by v_0 or v_1. If so, this information is recorded in *subst_table* where it is used by *build_result* to construct the final result.

The procedures *disjoin*, *conjoin* and *provably_empty* can be made approximate in a way that allows them to require only constant time and to still be accurate enough to give good simplification results. In this case, *simplify* requires time linear in the size of the DAG representing u.

Observability care sets have been used is logic synthesis [7, 11]. Since our full algorithm handles a subset of first-order logic, rather than just boolean netlists, it is an extension of these methods. Also, although observability care sets have been applied to formal verification [2], they have not previously been applied to logical transformations of symbolic expressions.

```
set of (expr $\times$ care_set) queue;
    /* priority queue for processing exprs in biggest-first */
    /* order, and for recording the care set of each expr */
set of (expr $\times$ expr) subst_table;
    /* hash table that maps an expr to a safe replacement */
set of (expr $\times$ expr) result_table;
    /* hash table that maps an expr to the expr it is */
    /* replaced by in the result, effectively the transitive */
    /* closure of subst_table */

expr simplify($u$, $b$);
    expr $u$;
    care_set $b$;
/* simplify $u$ using $b$ and using observability don't cares */
/* for subexprs of $u$ */
{
    expr $v$, $p$, $v_0$, $v_1$;
    care_set $c$, $c_0$, $c_1$;
    queue = $\emptyset$;
    subst_table = $\emptyset$;
    result_table = $\emptyset$;
    update_queue($u$, $b$);
    while (queue $\neq \emptyset$) {
        choose $(v, c) \in$ queue such that
            there does not exist $(v', c') \in$ queue
            where $v'$ is a proper subexpression of $v$;
        remove $(v, c)$ from queue;
        if ($v$ is of the form $ite(p, v_1, v_0)$) {
            $c_1$ = conjoin($c$, $p$);
            $c_0$ = conjoin($c$, $\neg p$);
            if (provably_empty($c_1$)) {
                    /* $p$ is replaceable by $false$ in $c$, so $v$ is */
                    /* replaceable by $v_0$ */
                    add $(v, v_0)$ to subst_table;
                    update_queue($v_0$, $c_0$);
                    }
                else {
                    update_queue($v_1$, $c_1$);
                    if (provably_empty($c_0$))
                        /* $p$ is replaceable by $true$ in $c$, so $v$ is */
                        /* replaceable by $v_1$ */
                        add $(v, v_1)$ to subst_table;
                    else {
                        update_queue($p$, $c$);
                        update_queue($v_0$, $c_0$);
                        };
                    };
            };
            /* else $v$ is atomic, so no further processing */
        };
    return build_result($u$);
    }
```

Figure 6: Simplification algorithm.

```
void update_queue($u$, $b$);
    expr $u$;
    care_set $b$;
/* update queue to reflect that $u$ must be processed and */
/* the care set of $u$ contains $b$ */
{
    care_set $c$, $c'$;
    if (($u$, $c$) $\in$ queue for some $c$) {
            remove $(u, c)$ from queue;
            $c'$ = disjoin($b$, $c$);
            add $(u, c')$ to queue;
            }
        else add $(u, b)$ to queue;
    }

expr build_result($u$);
    expr $u$;
/* use info in subst_table to build simplified $u$, */
/* memoize results in result_table */
{
    expr $v$, $w$, $p$, $u_0$, $u_1$;
    if (($u$, $w$) $\in$ result_table for some $w$)
        return $w$;
    if (($u$, $v$) $\in$ subst_table for some $v$) {
        $w$ = build_result($v$);
        add $(u, w)$ to result_table;
        return $w$;
        };
    if ($u$ is of the form $ite(p, u_1, u_0)$)
        $w$ = build_ite_expr(build_result($p$),
                    build_result($u_1$),
                    build_result($u_0$));
        else $w$ = $u$;
    add $(u, w)$ to result_table;
    return $w$;
    }
```

Figure 7: Routines used by the simplification algorithm.

6 Superscalar Example

In this section, we describe empirical results for applying our verification techniques to a pipelined, superscalar implementation of a subset of the DLX processor instruction set [8].

The subset of the DLX that we verified has the same six instruction classes as used by Burch and Dill [3]: store word, load word, unconditional jump, conditional branch (branch when the source register is equal to zero), 3-register ALU instructions, and ALU immediate instructions. The specifics of the ALU operations are abstracted away in both the specification and the implementation. Thus, our verification covers any set of ALU operations, assuming that the combinational ALU in the processor has been separately verified.

Instructions are loaded, two per cycle, into the instruction queue. If not enough instructions move out of the queue to make room for two instructions to be fetched, then no instructions are fetched. Up to nine instructions can be in the processor at one time.

The caches and the memory system are not modeled in any detail. Instead, instruction memory is treated as a black box: the processor outputs the PC and, after a non-deterministic delay, receives the instruction from the memory location corresponding to the PC. Modeling and verifying the interactions of the processor pipeline, caches and memory system is an area for future research.

The implementation uses a simple assume-branch-not-taken prediction strategy. As a result of the way it handles branches, this processor speculatively fetches and queues up instructions, but it does not do speculative execution.

Checking the commutative diagrams in figures 3(b) and 3(c) requires keeping the two sides of the diagrams "in sync" by having the user construct functions from implementation states to the integer variables i and m. For this processor, the desired function sets $i = 0$ if in the current state there is a branch taken or there is no room to sneak fetch an instruction; otherwise, $i = 1$. Similarly, $m = 0$ if in the current state there is a branch taken or no instructions are fetched; otherwise, $m = 2$. The user need not worry about the details of when branches are taken or instructions are fetched, this information can be derived by using the symbolic simulator to compute expressions for the appropriate control signals in the implementation. An error by the user or a bug in the processor can cause the resulting synchronization function to be wrong, but this can lead only to a false negative verification result, never a false positive.

If we directly checked the commutative diagram in figure 2, rather than the one in figure 3(b), then the synchronization function would have to be much more complicated. Instead of only having to consider a taken branch during the current cycle, it would have to handle instructions being squashed by branches potentially being taken during the next several cycles. Thus, the use of the commutative diagrams in figure 3 not only reduces CPU time needed for validity checking, it reduces manual effort, as well.

For this processor, scheduling the flushing of the implementation as described in section 4 requires adding two control inputs to the implementation. This gives us three modes for the processor: normal operation, allow no instructions to be issued, and allow at most one instruction to be issued. Using these modes, flushing the pipeline is scheduled as follows. First, allow no instructions to be issued for enough cycles that all previously issued instructions are guaranteed to be drained. Second, for one cycle, let one instruction be issued. Then, keep repeating these two steps until the processor is completely flushed. Following this schedule, rather than just flushing the processor without using the extra control inputs, produces much simpler expressions for the abstraction function.

When verifying this processor we made use of a number manually produced case splits. These were derived from a few simple concepts such the number of instructions in the queue or how many instructions were issued in a given cycle. They do not require an understanding of the expressions constructed when checking the commutative diagrams. These manual case splits were only needed for checking the commutative diagrams in figures 3(b) and 3(c); they were not needed for figure 3(a).

In summary, the verification procedure had the following steps. First, symbolically simulate the implementation and the specification descriptions to derive their transition functions. The user provides synchronization functions and a flushing schedule. Next, the verifier automatically uses the commutative diagrams to construct expressions that need to be checked for validity. A total of 28 manual case splits were used to simplify these expressions. Finally, the resulting expressions are processed by a simplification algorithm based on the one in section 5. For all of the expressions, the simplification algorithm returned the identically true expression. Thus, it was not necessary to use an automatic case splitting algorithm as in previous methods [3, 10]. The total CPU required for verification of this superscalar processor model was less than 30 minutes on a Sun 4.

7 Conclusion

We have extended the verification method of Burch and Dill [3] to enable the formal verification of a superscalar processor model against its instruction set architecture. Some of these extensions add to the manual effort that was already required with Burch and Dill's method. Thus, in some ways, our method is less automatic than many of the verification methods in the literature. However, our empirical results suggest that we have achieved an effective balance between automation and manual effort.

Of the three techniques described in this paper, the method for constructing abstraction functions (section 4) appears to have been the most important for verifying our processor model. In this model, from zero to two instructions can be issued per cycle and instructions wait in a fetch queue before being issued, so the latency of an instruction can vary significantly depending on the state of the pipeline when the instruction is fetched. Such variable latency is not present in other verified processors in the literature [1, 3, 4, 6, 9, 10, 12, 13], but it is a common property of modern commercial microprocessors. Without our technique for constructing abstraction functions, the Burch and Dill method [3] could not efficiently handle processors with such variable latency.

As described in the paper, errors in some of the manual inputs to the verification process can lead to false negative verification results, but never false positives. The impossibility of false positives is a consequence of well known properties of verification methods that use abstraction functions [1, 4, 6, 9, 12, 13]. If the user does make this kind of error, then the verification tool can be used to debug the user's inputs. Once the inputs are correct, then the verification and the debugging of the processor itself can begin.

REFERENCES

[1] D. L. Beatty. *A Methodology for Formal Hardware Verification, with Application to Microprocessors.* PhD thesis, School of Computer Science, Carnegie Mellon University, Aug. 1993.

[2] D. Brand. Verification of large synthesized designs. In *Intl. Conf. on Comp. Aided Design*, 1993.

[3] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In D. L. Dill, editor, *Conference on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1994.

[4] A. J. Cohn. A proof of correctness of the Viper microprocessors: The first level. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–72. Kluwer, 1988.

[5] F. Corella, M. Langevin, E. Cerny, Z. Zhou, and X. Song. State enumeration with abstract descriptions of state machines. In *Correct Hardware Design and Verification Methods, CHARME '95*, Oct. 1995.

[6] D. Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, Dec. 1993.

[7] M. Damiani and G. D. Micheli. Observability don't care sets and boolean relations. In *Intl. Conf. on Comp. Aided Design*, 1990.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

[9] W. A. Hunt, Jr. FM8501: A verified microprocessor. Technical Report 47, University of Texas at Austin, Institute for Computing Science, Dec. 1985.

[10] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *Intl. Conf. on Comp. Aided Design*, 1995.

[11] H. Savoj and R. K. Brayton. The use of observability and external don't cares for the simplification of multi-level networks. In *27th ACM/IEEE Design Automation Conference*, 1990.

[12] M. Srivas and S. P. Miller. Applying formal verification to a commercial microprocessor. In *Computer Hardware Description Languages*, Aug. 1995.

[13] P. J. Windley. Formal modeling and verification of microprocessors. *IEEE Trans. Comput.*, 44(1):54–72, Jan. 1995.