To appear in the Conference on Computer-Aided Verification, June 21-23, 1994
(this version has an appendix not present in the conference version)

Automatic Verification of Pipelined Microprocessor Control

Jerry R. Burch and David L. Dill

Computer Science Department
Stanford University

Abstract. We describe a technique for verifying the control logic of pipelined microprocessors. It handles more complicated designs, and requires less human intervention, than existing methods. The technique automatically compares a pipelined implementation to an architectural description. The CPU time needed for verification is independent of the data path width, the register file size, and the number of ALU operations. Debugging information is automatically produced for incorrect processor designs. Much of the power of the method results from an efficient validity checker for a logic of uninterpreted functions with equality. Empirical results include the verification of a pipelined implementation of a subset of the DLX architecture.

1 Introduction

The design of high-performance processors is a very expensive and competitive enterprise. The speed with which a design can be completed is a crucial factor in determining its success in the marketplace. Concern about design errors is a major factor in design time. For example, each month of additional design time of the MIPS 4000 processor was estimated to cost \$3-\$8 million, and 27% of the design time was spent in "verification and test" [13].

We believe that formal verification methods could eventually have a significant economic impact on microprocessor designs by providing faster methods for catching design errors, resulting in fewer design iterations and reduced simulation time. For maximum economic impact, a verification methodology should:

- be able to handle modern processor designs,
- be applicable to the aspects of the design that are most susceptible to errors,
- be relatively fast and require little labor, and
- provide information to help pinpoint design errors.

The best-known examples of formally verified processors have been extremely simple processor designs, which were generally unpipelined [7, 8, 15, 16]. The verification methods used rely on theorem-provers that require a great deal of very skilled human guidance (the practical unit of for measuring labor in these studies seems to be the person-month). Furthermore, the processor implementations that were verified were so simple that they were able to avoid central problems such as control complexity. There are more recent verification techniques [1, 17] that are much more automatic, but they have not been demonstrated on pipelined processors.

The verification of modern processors poses a special problem. The natural specification of a processor is the programmer-level functional model, called the *instruction set architecture.* Such a specification is essentially an operational description of a processor that executes each instruction separately, one cycle per instruction. The implementation, one the other hand, need not execute each instruction separately; several instruction might be executing simultaneously because of pipelining, etc. Formal verification requires proving that the specification and implementation are in a proper relationship, but that relationship is not necessarily easy to define.

Recently, there have been successful efforts to verify pipelined processors using human-guided theorem-provers [11, 19, 20, 22]. However, in all of these cases, either the processor was extremely simple or a large amount of labor was required.

Although the examples we have attacked are still much simpler than current high-performance commercial processors, they are significantly beyond the capabilities of automatic verification methods reported previously. The method is targeted towards errors in the microprocessor *control,* which, according to many designers, is where most of the bugs usually exist (datapaths are usually not considered difficult, except for floating point operations). Labor is minimized, since the procedure is automatic except for the development of the descriptions of the specification and implementation. When the implementation of the processor is incorrect, the method can produce a specific example showing how the specification is violated.

Since we wish to focus on the processor control, we assume that the combinational logic in the data path is correct. Under this assumption (which can be formally checked using existing techniques), the differences between the specification and implementation behaviors are entirely in the timing of operations and the transfer of values. For example, when the specification stores the sum of two registers in a destination register, the implementation may place the result in a pipe register, and not write the result to its destination until after another instruction has begun executing.

The logic we have chosen is the quantifier-free logic of uninterpreted functions and predicates with equality and propositional connectives. Uninterpreted functions are used to represent combinational ALUs, for example, without detailing their functionality. Propositional connectives and equality are used in describing control in the specification and the implementation, and in comparing them.

The validity problem for this logic is decidable. In practice, the complexity is dominated by handling Boolean connectives, just as with representations for propositional logic such as BDDs [2]. However, the additional expressiveness of our logic allows verification problems to be described at a higher level of abstraction than with propositional logic. As a result, there is a substantial reduction in the CPU time needed for verification.

Corella has also observed that uninterpreted functions and constants can be used to abstract away from the details of datapaths, in order to focus on control issues [9, 10]. He has a program for analyzing logical expressions which he has used for verifying a non-pipelined processor and a prefetch circuit. Although the details are not presented, his analysis procedure appears to be much less efficient than ours, and he does not address the problems of specifying pipelined processors.

Our method can be split into two phases. The first phase compiles operational descriptions of the specification and implementation, then constructs a logical formula that is valid if and only if the implementation is correct with respect to the specification. The second phase is a decision procedure that checks whether the formula is valid. The next two sections describe these two phases. We then give experimental results and concluding remarks.

2 Correctness Criteria

The verification process begins with the user providing behavioral descriptions of an implementation and a specification. For processor verification, the specification 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 takes a state as its first argument, the current inputs as its second argument, and returns the next state. The transition function 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_{Spec} and F_{Impl} to denote the transition function of the specification and the implementation, respectively. We require that the implementation and the specification have corresponding input wires. The processors we have verified have no explicit output wires since the memory was modeled as part of the processor and we did not model I/O.

Almost all processors have an input setting that causes instructions already in the pipeline to continue execution while no new instructions are initiated. This is typically referred to as *stalling* the processor. If I_{Stall} is an input combination that causes the processor to stall, then the function $F_{\mathrm{Impl}}(\,\cdot\,, I_{\mathrm{Stall}})$ represents the effect of stalling for one cycle. All instructions currently in the pipeline can be completed by stalling for a sufficent number of cycles. This operation is called *flushing* the pipeline, and it is an important part of our verification method.

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. The verifier solves this problem by simulating the effect of completing every instruction in the pipeline before doing the comparison. The natural way to complete every instruction is to flush the pipeline.

All of this is made more precise in figure 1. The implementation can be in an arbitrary state Q_{Impl} (labeled "Old Impl State" in the figure). To complete the partially executed instructions in Q_{Impl}, the pipeline is flushed, producing "Flushed Old Impl State". Then, all but the programmer-visible parts of the implementation state are stripped off (we define the function *proj* for this purpose) to produce Q_{Spec}, the "Old Spec State". Because of the way Q_{Spec} is constructed from Q_{Impl}, we say that Q_{Spec} *matches* Q_{Impl}.

$$\begin{tikzcd}
\boxed{\text{Old Impl State}} \arrow[r, "F_{\text{Impl}}(\,\cdot\,,I_{\text{Stall}})"] \arrow[d, "F_{\text{Impl}}(\,\cdot\,,I)"] & \bullet\bullet\bullet \arrow[r, "F_{\text{Impl}}(\,\cdot\,,I_{\text{Stall}})"] & \boxed{\text{Flushed Old Impl State}} \arrow[r, "proj"] & \boxed{\text{Old Spec State}} \arrow[d, "F_{\text{Spec}}(\,\cdot\,,I)"'] \\
\boxed{\text{New Impl State}} \arrow[r, "F_{\text{Impl}}(\,\cdot\,,I_{\text{Stall}})"'] & \bullet\bullet\bullet \arrow[r, "F_{\text{Impl}}(\,\cdot\,,I_{\text{Stall}})"'] & \boxed{\text{Flushed New Impl State}} \arrow[r, "proj"'] & \boxed{\text{New Spec State}}
\end{tikzcd}$$

Fig. 1. Commutative diagram for showing our correctness criteria.

Let I be an arbitrary input combination to the pipeline (recall that the specification and the implementation are required to have corresponding input wires). Let $Q'_{\text{Impl}} = F_{\text{Impl}}(Q_{\text{Impl}}, I)$, the "New Impl State", and let $Q'_{\text{Spec}} = F_{\text{Spec}}(Q_{\text{Spec}}, I)$, the "New Spec State". We consider the implementation to satisfy the specification if and only if Q'_{Spec} matches Q'_{Impl}. To check this, flush and project Q'_{Impl}, then see if the result is equal to Q'_{Spec}, as shown at the bottom of figure 1.

It is often convenient to use a slightly different (but equivalent) statement of our correctness criteria. In figure 1, there are two different paths from "Old Impl State" to "New Spec State". The path that involves $F_{\text{Impl}}(\,\cdot\,, I)$ is called the *implementation side* of the diagram; the path that involves $F_{\text{Spec}}(\,\cdot\,, I)$ is called the *specification side.* For each path, there is a corresponding function that is the composition of the functions labeling the arrows on the path. We say that the implementation satisfies the specification if and only if the function corresponding to the implementation side of the diagram is equal to the function corresponding to the specification side of the diagram. More succinctly, the diagram must *commute.*

The reader may notice that figure 1 has the same form as commutative diagrams used with *abstraction functions.* In our case, the abstraction function represents the effect of flushing an implementation state and then applying the *proj* function. Typical verification methods require that there exist some abstraction function that makes the diagram commute. In contrast, we require that our specific abstraction function makes the diagram commute.

In some cases, it may be necessary during verification to restrict the set of "Old Impl States" considered in figure 1. In this case, an *invariant* could be provided (the invariant must be closed under the implementation transition function). All of the examples in this paper were proved correct without having to use an invariant.

Notice that the same input I is applied to both the implementation and the specification in figure 1. This is only appropriate in the simple case where the implementation requires exactly one cycle per instruction (once the pipeline is filled). If more than one cycle is sometimes required, then on the extra cycles it is necessary to apply I_{Stall} to the inputs of the specification rather than I. An example of this is discussed in section 4.2.

3 Checking Correctness

As described above, to verify a processor we must check whether the two functions corresponding to the two sides of the diagram in figure 1 are equal. Each of the two functions can be represented by a vector of symbolic expressions. The vectors have one component for each programmer-visible state variable of the processor. These expressions can be computed efficiently by symbolically simulating the behavioral descriptions of the implementation and the specification. The implementation is symbolically simulated several times to model the effect of flushing the pipeline.

Let $\langle s_1, \ldots, s_n \rangle$ and $\langle t_1, \ldots, t_n \rangle$ be vectors of expressions. To verify that the functions they represent are equal, we must check whether each formula $s_k = t_k$ is valid, for $1 \leq k \leq n$. Before describing our algorithm for this, we define the logic we use to encode the formulas.

3.1 Uninterpreted Functions with Equality

Many quantifier-free logics that include uninterpreted functions and equality have been studied. Unlike most of those logics [18, 21], ours does not include addition or any arithmetical relations. For our application of verifying microprocessor control, there does not appear to be any need to have arithmetic built into the logic (although the ability to declare certain uninterpreted functions to be associative and/or commutative would be useful).

We begin by describing a subset of the logic we use. This subset has the following abstract syntax (where *ite* denotes the if-then-else operator):

$$\begin{aligned}
\langle \textit{formula} \rangle ::= \;& \textit{ite}(\langle \textit{formula} \rangle,\, \langle \textit{formula} \rangle,\, \langle \textit{formula} \rangle) \\
& \mid (\langle \textit{term} \rangle = \langle \textit{term} \rangle) \\
& \mid \langle \textit{predicate symbol} \rangle(\langle \textit{term} \rangle, \ldots, \langle \textit{term} \rangle) \\
& \mid \langle \textit{propositional variable} \rangle \mid \textit{true} \mid \textit{false}
\end{aligned}$$

$$\begin{aligned}
\langle \textit{term} \rangle ::= \;& \textit{ite}(\langle \textit{formula} \rangle,\, \langle \textit{term} \rangle,\, \langle \textit{term} \rangle) \\
& \mid \langle \textit{function symbol} \rangle(\langle \textit{term} \rangle, \ldots, \langle \textit{term} \rangle) \\
& \mid \langle \textit{term variable} \rangle.
\end{aligned}$$

Notice that the *ite* operator can be used to construct both formulas and terms. We included *ite* as a primitive because it simplifies our case-splitting heuristics and because it allows for efficient construction of transition functions without introducing auxiliary variables.

There is no explicit quantification in the logic. Also, we do not require specific interpretations for function symbols and predicate symbols. A formula is valid if and only if it is true for all interpretations of variables, function symbols and predicate symbols.

Although the *ite* operator, together with the constants *true* and *false*, is adequate for constructing all Boolean operations, we also include logical negation and disjunction as primitives in our decision procedure. This simplifies the rewrite rules used to reduce our formulas, especially rules involving associativity and commutativity of disjunction.

Verifying a processor usually requires reasoning about *stores* such as a register file or main memory. We model stores as having an unbounded address space. If a processor design satisfies our correctness criteria in this case, then it is correct for any finite register file or memory. If certain conventions are followed, the above logic is adequate for reasoning about stores. However, we found it more efficient to add two primitives, *read* and *write*, for manipulating stores. These primitives are essentially the same as the *select* and *store* operators used by Nelson and Oppen [18]. If *regfile* is a variable representing the initial state of a register file, then

$$\textit{write}(\textit{regfile},\, \textit{addr},\, \textit{data})$$

represents the store that results from writing the value *data* into address *addr* of *regfile*. The value at address *addr* in the original state of the register file is denoted by

$$\textit{read}(\textit{regfile},\, \textit{addr}).$$

Any expression that denotes a store, whether it is constructed using variables, *write*'s or *ite*'s, can be used as the first argument of a *read* or a *write* operation.

3.2 Validity Checking Algorithm

Pseudo-code for a simplified version of our decision procedure for checking validity, along with a description of its basic operation, is given in figure 2. This procedure is still preliminary and may be improved further, so we will just sketch the main ideas behind it.

Our decision procedure differs in several respects from earlier work [18, 21]. Arithmetic is not a source of complexity for our algorithm, since it is not included in our logic. In our applications, the potentially complex Boolean structure of the formulas we check is the primary bottleneck. Thus, we have concentrated on handling Boolean structure efficiently in practice.

Another difference is that we are careful to represent formulas as directed acyclic graphs (DAGs) with no distinct isomorphic subgraphs. For this to reduce the time complexity of the validity checker, it is necessary to memoize (cache) intermediate results. As shown in figure 2, the caching scheme is more complicated than in standard BDD algorithms [2] because formulas must be cached relative to a set of assumptions.

The final major difference between our algorithm and previous work is that we do not require formulas to be rewritten into a Boolean combination of atomic formulas. For example, formulas of the form $e_1 = e_2$, where e_1 and e_2 may contain an arbitrary number of *ite* operators, are checked directly without first being rewritten.

Detlefs and Nelson [12] have recently developed a new decision procedure based on a conjunctive normal form representation that appears to be efficient in practice. We have not yet been able to do a thorough comparison, however.

As *check* does recursive case analysis on the formula p, it accumulates a set of assumptions A that is used as an argument to deeper recursive calls (see figure 2). This set of assumptions must not become inconsistent. To avoid such inconsistency, we require that if p_0 is the first result of $\mathit{simplify}(p, A_0)$, then neither $\mathit{choose\_splitting\_formula}(p_0)$ nor its negation is logically implied by A_0. We call this the *consistency requirement* on *simplify* and *choose_splitting_formula*. Maintaining the consistency requirement is made easier by restricting the procedure *choose_splitting_formula* to return only atomic formulas (formulas containing no *ite*, *or*, *not* or *write* operations).

As written in figure 2, our algorithm is indistinguishable from a propositional tautology checker. Dealing with uninterpreted functions and equality is not done in the top level algorithm. Instead, it is done in the *simplify* routine. For example, given the assumptions $e_1 = e_2$ and $e_2 = e_3$, it must be possible to simplify the formula $e_1 \neq e_3$ to *false*; otherwise, the consistency requirement would not be maintained. As a result, *simplify* and associated routines require a large fraction of the code in our verifier.

In spite of the consistency requirement, there is significant latitude in how aggressively formulas are simplified. It may seem best to do as much simplification as possible, but our experiments indicate otherwise. We see two reasons for this. If *simplify* does the minimal amount of simplification necessary to meet the consistency requirement, then it may use less CPU time than a more aggres-

```
check($p$: formula, $A$: set of formula): set of formula;
    var
        $s$, $p_0$, $p_1$: formula;
        $A_0$, $A_1$, $U_0$, $U_1$, $U$: set of formula;
        $\mathcal{G}$: set of set of formula;
    begin
        if $p \equiv \mathit{true}$ then return $\emptyset$;
        if $p \equiv \mathit{false}$ then
            print "not valid";
            terminate unsuccessfully;
        $\mathcal{G}$ := cache($p$);
        if $\exists U \in \mathcal{G}$ such that $U \subseteq A$ then return $U$;      /* cache hit */
        /* cache miss */
        $s$ := choose_splitting_formula($p$);      /* prepare to case split */
        /* do $s$ false case */
        $A_0 := A \cup \{\neg s\}$;
        ($p_0$, $U_0$) := simplify($p$, $A_0$);
        $U_0 := U_0 \cup$check($p_0$, $A_0$);      /* assumptions used for $s$ false case */
        /* do $s$ true case */
        $A_1 := A \cup \{s\}$;
        ($p_1$, $U_1$) := simplify($p$, $A_1$);
        $U_1 := U_1 \cup$check($p_1$, $A_1$);      /* assumptions used for $s$ true case */
        $U := (U_0 - \{\neg s\}) \cup (U_1 - \{s\})$;      /* assumptions used */
        cache($p$) := $\mathcal{G} \cup \{U\}$;      /* add cache entry */
        return $U$;
    end;
```

Fig. 2. The procedure *check* terminates successfully if the formula p is logically implied by the set of assumptions A; otherwise, it terminates unsuccessfully. Not all of the assumptions in A need be relevant in implying p; when *check* terminates successfully it returns those assumptions that were actually used. The set of assumptions used need not be one of the minimal subsets of A that implies p. Checking whether p is valid is done by letting A be the emptyset. Initially, the global lookup table *cache* returns the emptyset for every formula p. Later, *cache*(p) returns the set containing those assumption sets that have been sufficient to imply p in previous calls of *check.* The procedure *choose_splitting_formula* heuristically chooses a formula to be used for case splitting. The call *simplify*(p, A_0) returns as its first result a formula formed by simplifying p under the assumptions A_0. The second result is the set of formulas in A_0 that were actually used when simplifying p.

sive simplification routine. Thus, even if slightly more case splitting is needed (resulting in more calls to *simplify*), the total CPU time used may be reduced.

The second reason is more subtle. Suppose we are checking the validity of a formula p that has a lot of shared structure when represented as a DAG. Our hope is that by caching intermediate results, the CPU time typically needed for validity checking grows with the size of the DAG of p, rather than with the size of its tree representation. This can be important in practice; for the DLX example (section 4.2) it is not unusual for the tree representation of a formula to be two orders of magnitude larger than the DAG representation. The more aggressive *simplify* is, the more the shared structure of p is lost during recursive case analysis, which appears to result in worse cache performance. We are continuing to experiment with different kinds of simplification strategies in our prototype implementation.

Unlike the algorithm in figure 2, our validity checker produces debugging information for invalid formulas. This consists of a satisfiable set of (possibly negated) atomic formulas that implies the negation of the original formula. When verifying a microprocessor, the debugging information can be used to construct a simulation vector that demonstrates the bug.

There is another important difference between our current implementation and the algorithm in figure 2. Let (p_0, U_0) be the result of *simplify*(p, A_0). Contrary to the description in figure 2, in our implementation U_0 is not required to be a subset of A_0. All that is required is that all of the formulas in U_0 are logically implied by A_0, and that the equivalence of p and p_0 is logically implied by U_0. As a result, something more sophisticated than subtracting out the sets $\{s\}$ and $\{\neg s\}$ must be done to compute a U that is weak enough to be logically implied by A (see figure 2). A second complication is that finding a cache hit requires checking sufficient conditions for logical implication between sets of formulas, rather than just set containment. However, dealing with these complications seems to be justified since the cache hit ratio is increased by having *simplify* return a U_0 that is weaker than it could be if it had to be a subset of A_0. We are still experimenting with ideas for balancing these issues more efficiently.

4 Experimental Results

In this section, we describe empirical results for applying our verification method to a pipelined ALU [5] and a subset of the DLX processor [14].

4.1 Pipelined ALU

The 3-stage pipelined ALU we considered (figure 3) has been used as a benchmark for BDD-based verification methods [3, 4, 5, 6]. A natural way to compare the performance of these methods is to see how the CPU time needed for verification grows as the pipeline is increased in size by (for example) increasing its datapath width w or its register file size r. For Burch, Clarke and Long [4] the CPU time grew roughly quadratically in w and cubically in r. Clarke, Grumberg and Long [6], using a simple abstraction provided by the user, demonstrated linear growth in both w and r. Sublinear growth in r and subquadratic growth in w was achieved by Bryant, Beatty and Seger [3].

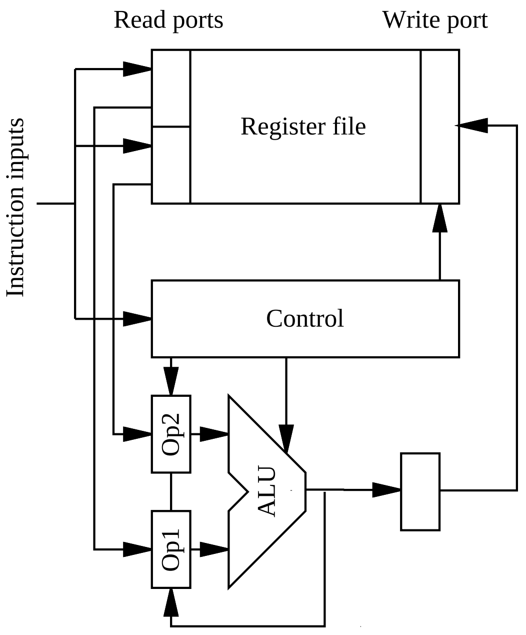

Fig. 3. 3-stage pipelined ALU. If the *stall* input is true, then no instruction is loaded. Otherwise, the *src1* and *src2* inputs provide the address of the arguments in the register file, the *op* input specifies the ALU operation to be performed on the arguments, and the *dest* input specifies were the result is to be written.

In our verification method, the width of the data path and the number of registers and ALU operations can be abstracted away. As a result, one verification run can check the control logic of pipelines with any combination of values for these parameters. A total of 370 milliseconds of CPU time (running compiled Lucid Common LISP on a DECstation 5000/240) is required to do a complete verification run, including loading and compiling behavioral descriptions, automatically constructing the abstraction function and related expressions, and checking the validity of the appropriate formula. The validity checking itself, the primary bottleneck on larger examples, only required 50 milliseconds for the pipelined ALU.

4.2 DLX Processor

Hennessy and Patterson [14] designed the DLX architecture to teach the basic concepts used in the MIPS 2000 and other RISC processors of that generation.

The subset of the DLX that we verified had six types of instructions: 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. As with the pipelined ALU described earlier, 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.

Our DLX implementation has a standard 5-stage pipeline. The DLX architecture has no branch delay slot; our implementation uses the "assume branch not taken" strategy. No pipelining is exposed in the DLX architecture or in our specification of it. Thus, it is the responsibility of the implementation to provide forwarding of data and a load interlock.

The interlock and the lack of a branch delay slot mean that the pipeline executes slightly less than one instruction per cycle, on average. This complicates "synchronizing" the implementation and the specification during verification, since the specification executes exactly one instruction per cycle. We address the problem in a manner similar to that used by Saxe *et al.* [20]. The user must provide a predicate on the implementation states that indicates whether the instruction to be loaded on the current cycle will actually be executed by the pipeline. While this predicate can be quite complicated, it is easy to express in our context, using internal signals generated by the implementation. In particular, our pipeline will not execute the current instruction if and only if one or more of the following conditions holds: the stall input is asserted, the signal indicating a taken branch is asserted, or the signal indicating that the pipeline has been stalled by the load interlock is asserted.

When internal signals are used in this way, it is possible for bugs in the pipeline to lead to a false positive verification result. In particular, the pipeline may appear correct even if it can get into a state where it refuses to ever execute another instruction (a kind of livelock). To avoid the possibility of a false positive, we automatically check a progress condition that insures that livelock cannot occur. The CPU time needed for this check is included in the total given below.

Our specification has four state variables: the program counter, the register file, the data memory and the instruction memory. If the data memory and the instruction memory are combined into one store in the specification and the implementation, then the verifier will detect that the pipeline does not satisfy the specification for certain types of self-modifying code (this has been confirmed experimentally). Separating the two stores is one way to avoid this inappropriate negative result.

For each state variable of the specification, the verifier constructs an appropriate formula and checks its validity. Since neither the specification nor the implementation write to the instruction memory, checking the validity of the corresponding formula is trivial. Checking the formulas for the program counter, the data memory and the register file requires 15.5 seconds, 34 seconds and 9.5 seconds of CPU time, respectively. The total CPU time required for the full verification (including loading and compiling the behavioral descriptions, etc.)

is less than 66 seconds.

In another test, we introduced a bug in the forwarding logic of the pipeline. The verifier required about 8 seconds to generate 3 counter-examples, one each for the three formulas that had to be checked. These counter-examples provided sufficient conditions on a initial implementation state where the effects of the bug would be apparent. This information can be analyzed by hand, or used to construct a start state for a simulator run that would expose the bug.

5 Concluding Remarks

The need for improved debugging tools is now obvious to everyone involved in producing a new processor implementation. It is equally obvious that the problem is worsening rapidly: driven by changes in semiconductor technology, architectures are moving steadily from the simple RISC machines of the 1980s towards very complex machines which aggressively exploit concurrency for greater performance.

Although we have demonstrated that the techniques presented here can verify more complex processors with much less effort than previous work, examples such as our DLX implementation are still not nearly as complex as commercial microprocessor designs. We have also not yet dealt with memory systems and interrupts, which are rich source of bugs in practice.

It will be very challenging to increase the capacity of verification tools as quickly as designers are increasing the scale of the problem. Clearly, the computational efficiency of logical decision procedures (in practice, not in the worst case) will be a major bottleneck. If decision procedures cannot be extended rapidly enough, it may still be possible to use some of the same techniques for partial verification or in a mixed simulation/verification tool.

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. K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *27th ACM/IEEE Design Automation Conference*, 1990.
3. R. E. Bryant, D. L. Beatty, and C.-J. H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *28th ACM/IEEE Design Automation Conference*, 1991.
4. J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *28th ACM/IEEE Design Automation Conference*, 1991.
5. J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, 1990.

6. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Nineteenth Annual ACM Symposium on Principles on Programming Languages*, 1992.
7. 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.
8. A. J. Cohn. Correctness properties of the Viper block model: The second level. In G. Birtwistle, editor, *Proceedings of the 1988 Design Verification Conference*. Springer-Verlag, 1989. Also published as University of Cambridge Computer Laboratory Technical Report No. 134.
9. F. Corella. Automated high-level verification against clocked algorithmic specifications. Technical Report RC 18506, IBM Research Division, Nov. 1992.
10. F. Corella. Automatic high-level verification against clocked algorithmic specifications. In *Proceedings of the IFIP WG10.2 Conference on Computer Hardware Description Languages and their Applications*, Ottawa, Canada, Apr. 1993. Elsevier Science Publishers B.V.
11. D. Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, Dec. 1993.
12. D. Detlefs and G. Nelson. Personal communication, 1994.
13. J. L. Hennessy. Designing a computer as a microprocessor: Experience and lessons from the MIPS 4000. A lecture at the Symposium on Integrated Systems, Seattle, Washington, March 14, 1993.
14. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.
15. W. A. Hunt, Jr. FM8501: A verified microprocessor. Technical Report 47, University of Texas at Austin, Institute for Computing Science, Dec. 1985.
16. J. Joyce, G. Birtwistle, and M. Gordon. Proving a computer correct in higher order logic. Technical Report 100, Computer Lab., University of Cambridge, 1986.
17. M. Langevin and E. Cerny. Verification of processor-like circuits. In P. Prinetto and P. Camurati, editors, *Advanced Research Workshop on Correct Hardware Design Methodologies*, June 1991.
18. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Prog. Lang. Syst.*, 1(2):245–257, Oct. 1979.
19. A. W. Roscoe. Occam in the specification and verification of microprocessors. *Philosophical Transactions of the Royal Society of London, Series A: Physical Sciences and Engineering*, 339(1652):137–151, Apr. 15, 1992.
20. J. B. Saxe, S. J. Garland, J. V. Guttag, and J. J. Horning. Using transformations and verification in circuit design. Technical Report 78, DEC Systems Research Center, Sept. 1991.
21. R. E. Shostak. A practical decision procedure for arithmetic with function symbols. *J. ACM*, 26(2):351–360, Apr. 1979.
22. M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, Sept. 1990.

A Illustrative Example

In this appendix, we give a more detailed description of how we verified the pipelined ALU in figure 3. The user must provide behavioral descriptions of the specification and the implementation (figures 4 and 5). The pseudo-code in figures 4 and 5 is generated by simple syntactic transformations from the LISP-like description language that is used with our prototype verifier. Notice that the pipelining of the ALU is completely abstracted away in the specification, but is quite explicit in the description of the implementation.

The two behavioral descriptions are automatically compiled into transition functions (figures 6 and 7). The transition functions are encoded by associating each state variable with an expression that represents how that state variable is updated each cycle. The expressions are in the logic of uninterpreted functions and equality described earlier. The concrete syntax for expressions in figures 6 through 10 is that used in our prototype verifier, which is implemented in LISP.

The verifier automatically produces an expression for the appropriate abstraction function (figure 8). The abstraction function and the transition functions are used to automatically produce expressions (figures 9 and 10) that correspond to the two sides of the commutative diagram in figure 1. The implementation is correct if and only if these expressions are equivalent, which can be automatically verified using the validity checker described earlier.

```
if stall
  then regfile' := regfile;
  else
    regfile' :=
      write(regfile, dest,
             alu(op, read(regfile, src1)
                     read(regfile, src2)));
```

Fig. 4. Specification for the pipelined ALU in figure 3. Primed variables refer to next state values; unprimed variables refer to current state values. Notice that pipelining is not apparent at the level of abstraction of the specification.

```
if bubble-wb
  then regfile' := regfile;
  else regfile' := write(regfile, dest-wb, result);

bubble-wb' := bubble-ex;
dest-wb' := dest-ex;
result' := alu(op-ex, arg1, arg2);

bubble-ex' := stall;
dest-ex' := dest;
op-ex' := op;
if (not bubble-ex) and (dest-ex = src1)
  then arg1' := result';
  else arg1' := read(regfile', src1);
if (not bubble-ex) and (dest-ex = src2)
  then arg2' := result';
  else arg2' := read(regfile', src2);
```

Fig. 5. Implementation of the pipelined ALU. The first three lines of code specify how the register file is updated by the final ("write back" or "wb") stage of the pipeline. The next three lines specify how three pipe registers are updated by the second ("execute" or "ex") stage. The remaining code specifies how five additional pipe registers are updated by the first stage of the pipeline. Notice that *arg1* is updated with the next state (rather than the current state) value of either *result* or *regfile;* this is necessary for data forwarding to work properly in the pipeline.

```
regfile:  (ite stall
               regfile
               (write regfile
                 dest
                 (alu op
                      (read regfile src1)
                      (read regfile src2))))
```

Fig. 6. Since the register file is the only state variable in the specification, the transition function has only one component, which is represented by the above expression. This expression is automatically compiled from a LISP-like version of the pseudo-code in figure 4.

```
regfile:  (ite bubble-wb
               regfile
               (write regfile dest-wb result))

arg1:  (ite (or bubble-ex
                (not (= src1 dest-ex)))
            (read
              (ite bubble-wb
                   regfile
                   (write regfile dest-wb result))
              src1)
            (alu op-ex arg1 arg2))
```

Fig. 7. The transition function for the pipeline has nine components, two of which are represented by the above expressions. These expressions are automatically compiled from a LISP-like version of the pseudo-code in figure 5.

```
(ite bubble-ex
     #1=(ite bubble-wb
             regfile
             (write regfile dest-wb result))
     (write #1#
            dest-ex
            (alu op-ex arg1 arg2)))
```

Fig. 8. This expression represents the abstraction function from a pipeline state to a specification state (only one expression is necessary since the register file is the only component of a specification state). The "#1#" notation denotes the use of a subterm that was defined earlier in the expression by the "#1=" notation. This expression is computed automatically from the transition function of the pipeline; the user need only specify how many times to iterate the transition function to flush the pipeline (two times, for this example).

```
(ite stall
     #1=(ite bubble-ex
             #2=(ite bubble-wb
                     regfile
                     (write regfile dest-wb result))
             (write #2#
               dest-ex
               (alu op-ex arg1 arg2)))
     (write #1#
       dest
       (alu op
            (read #1# src1)
            (read #1# src2))))
```

Fig. 9. This expression represents the functional composition of the abstraction function and the transition function of the specification. The verifier can compute this composition automatically. The resulting function corresponds to the specification side of the commutative diagram in figure 1.

```
(ite stall
     #1=(ite bubble-ex
             #2=(ite bubble-wb
                     regfile
                     (write regfile dest-wb result))
             (write #2#
               dest-ex
               #3=(alu op-ex arg1 arg2)))
     (write #1#
       dest
       (alu op
            (ite (or bubble-ex
                     (not (= src1 dest-ex)))
                 (read #2# src1)
                 #3#)
            (ite (or bubble-ex
                     (not (= dest-ex src2)))
                 (read #2# src2)
                 #3#))))
```

Fig. 10. This expression represents the functional composition of the transition function of the implementation and the abstraction function. The verifier can compute this composition automatically. The resulting function corresponds to the implementation side of the commutative diagram in figure 1.