Efficient Validity Checking for Processor Verification *

Robert B. Jones , David L. Dill
Computer Systems Laboratory, Stanford University, Stanford, CA 94305

Jerry R. Burch
Cadence Berkeley Laboratories, Berkeley, CA 94704

Abstract

We describe an efficient validity checker for the quantifier-free logic of equality with uninterpreted functions. This logic is well suited for verifying microprocessor control circuitry since it allows the abstraction of datapath values and operations. Our validity checker uses special data structures to speed up case splitting, and powerful heuristics to reduce the number of case splits needed. In addition, we present experimental results and show that this implementation has enabled the automatic verification of an actual high-level microprocessor description.

1 Introduction

As microprocessor designs become more complex, the cost of validation becomes a larger fraction of the total design cost. Currently, validation consumes 25-30% of the design team and months of simulation time. Industry experts predict that there may soon be two or three validation engineers for every design engineer on major microprocessor design projects [Wil95].

Although today's theorem provers could be used, in theory, to formally verify modern processors, the time and expertise required would be prohibitively expensive (indeed, such an effort might *increase* the length of the design cycle). Advances in BDD-based verification methods are not closing the tremendous gap in complexity between modern commercial microprocessor designs and those designs that can be automatically verified.

Burch and Dill [BD94b] proposed a new method for verifying microprocessor control circuitry. The method is based on a subset of first-order logic, specifically, the quantifier-free logic of equality with uninterpreted functions. This logic is appropriate for verification of microprocessor control because it allows abstraction of datapath values and operations. By contrast, propositional logic requires that individual bits be modeled explicitly.

Burch and Dill's verification method has two phases. The first phase compiles a behavioral description of the specification and implementation into a formula in the logic; if this formula is valid then the implementation is correct with respect to the specification. The second phase is a program that checks whether the formula is valid.

In this paper, we concentrate on the second phase of the method: *validity checking.* We describe fast, compact data structures that significantly speed up the inner loop of the validity checker. Our experiments also explore the trade-offs of using several different heuristics.

The logic studied here is a fundamental building-block for processor verification, and is useful for a variety of different verification methods. We believe that decidable logics which are more expressive than propositional logic (and programs to manipulate them) are going to be very important for verification as well as other CAD applications. Logics with uninterpreted functions are especially interesting, since the separation between control and datapath is fundamental to many design methods.

Although there have been many previous processor verification efforts, we discuss only those that are highly automated and have been applied to relatively complex designs. We believe that our method can deal with more complex designs than these previous efforts. For example, Beatty [Bea93] verified a switch-level non-pipelined processor description by using BDDs and symbolic simulation. In general, pipelining greatly increases the difficulty of the verification problem, so it is unclear whether Beatty's method, or other BDD-based methods, could cope with our design examples.

Although Bhagwati and Devadas [BD94a] claimed to verify a pipelined implementation of the DLX processor architecture (one of our examples) using BDDs and symbolic simulation, their method relies on the simplifying assumption that the pipelined implementation is k-definite. Although a correctly functioning pipeline may satisfy this assumption, design errors can result in behaviors that are not k-definite. Since the assumption is not checked, their method can miss a class of bugs that is of both theoretical and practical importance. We conjecture that eliminating the assumption k-definiteness in this method would make verification of the pipelined DLX design infeasible.

Corella *et al.* [C$^+$ 94] describe a canonical-form representation for expressions in a subset of first-order logic which is somewhat similar to ours. The method is based on iteratively computing a symbolic representation of the reachable states of the system, similar to BDD-based verifiers. However, iteration is generally much more difficult than symbolically simulating a small, fixed number of steps, so we believe that this method will not be able to handle large designs. Furthermore, the expressiveness of their logical representation is unclear.

Decision procedures for larger subsets of first order logic have been studied by several others, generally in the context of more general theorem-proving systems. Nelson and Oppen [NO79] give a decision procedure for the quantifier-free theory of the real numbers under $+$ and $\leq$, arrays, list structure, and equality with uninterpreted function symbols. Later extensions included congruence closure [NO80]. Shostak [Sho79] also implemented a general decision procedure for a quantifier-free logic richer than ours. These extensions to our subset of first-order logic are not necessary for verifying processor control. By using a more restricted logic, we can construct a faster validity checker.

There is an extensive literature on using general purpose theorem provers to verify processor designs, including recent work on verifying pipelined processors [Cyr93, SB90, SM95, Win95]. These methods require significantly more manual effort than our technique.

2 The logic

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

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

* This research was partially supported by the Semiconductor Research Corporation under contract number 94-DJ-389. The first author is supported by a National Defense Science and Engineering Graduate Fellowship.

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

Our logic has the following abstract syntax:

formula ::= **ite** (*formula*, *formula*, *formula*)
| (*term* = *term*)
| *predicate symbol* (*term*, . . ., *term*)
| **true** | **false**
term ::= **ite** (*formula*, *term*, *term*)
| *function symbol* (*term*, . . ., *term*)
| **read** (*term*, *term*)
| **write** (*term*, *term*, *term*)
| *distinct constant.*

A function of no arguments is a variable, which can be written without the following parentheses. *ite* represents the if-then-else operator, which may appear as a formula (returning a Boolean value), or as a term. The *ite* operator together with the truth constants *true* and *false* is sufficient for representing all Boolean operators. The parser for our implementation macro-expands *and*, *or*, etc. into equivalent expressions in the above syntax.

Distinct constants are automatically assumed not to be equal unless they are identical. This feature is useful in processor verification, for example to represent distinct instruction opcodes. In this paper, distinct constants are distinguished with a leading "@"; for example, "@a" is a distinct constant.

It is helpful when verifying processors to be able to reason about *stores* (memories) such as register files, caches, or main memory. Formally, a store is a function of one argument (the *address*). There is special support for stores in the logic, in the form of two special operations: *read* and *write* (similar to the *select* and *store* operators used by Nelson and Oppen [NO79]). The expression *read*$(store, addr)$ is the value at address *addr* of store *store.* The expression *write*$(store, addr, val)$ is the store that has the value *val* at address *addr*, and the same values as *store* for all other addresses.

Note that the logic's view 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.

An expression in the logic is said to be *atomic* if it does not contain any *ite* or *write* operations.

3 Validity checking

The symbolic simulation step of the verification procedure generates a logical expression which should be *valid*, meaning that it is true under every interpretation. Checking for validity, in essence, covers all of the cases that must be analyzed to ensure that the processor works for every possible instruction sequence and initial state.

The validity checking problem is a generalization of the *tautology checking* problem for propositional formulas. However, the validity checking problem for our logic is more difficult, because it must take into account additional properties of equality and functions. For example, if it is known that $a = b$ and $b = c$, then it is known that $a = c$ and that $f(a) = f(c)$.

3.1 Propositional case

First, let us consider a straightforward validity checker, based on Shannon decomposition, for propositional formulas (the subset of our logic that does not have equality, function symbols, or predicate symbols). The algorithm of Figure 1 can be used to check such formulas, once certain functions are explained. Indeed, many tautology checkers for propositional formulas are based on this procedure [LCDM89].

For propositional formulas, a *context* is a truth assignment to a subset of the propositional variables. The atomic formulas in our logic are equivalent to propositional variables. An atomic formula can be *asserted* in a context (which assigns a propositional variable

```
CheckValidity(Expr e)
    Expr splitter;
    if (IsAtomic(e))
        if (Fold(e) ≠ True)
            print("Invalid result.");
            PrintContext(TopScope);
            Debug();
    else
        splitter = FindSplitter(e);
        PushContext();
        Assert (splitter);
        CheckValidity (Fold(e));
        PopContext();
        PushContext();
        Deny (splitter);
        CheckValidity (Fold(e));
        PopContext();
```

Figure 1: Validity Checking Algorithm.

the value *true*), or *denied* (assigned *false*). A context may imply a given atomic formula (if the variable is assigned *true* in the context), imply that the atomic formula is false (if it is assigned *false*), or not determine the truth of the atomic formula.

The *Fold* function simplifies an expression by replacing each atomic formula by its value in the current context, if the context determines the value; otherwise, the atomic formula is not changed. In addition, transformations such as:

$$ite(true, \alpha, \beta) \Longrightarrow \alpha,$$
$$ite(\alpha, \beta, \beta) \Longrightarrow \beta,$$

are performed. The formula returned by *Fold* is guaranteed to contain no atomic formulas that are determined by the context in which the folding occurred.

Consider the following simple example, in a context where $a = true$ and $b = false$:

$$ite\ (ite\ (a, b, c), d, e) \Longrightarrow$$
$$ite\ (ite\ (true, false, c), d, e) \Longrightarrow$$
$$ite\ (false, d, e) \Longrightarrow e.$$

PushContext creates a copy of the current context which can be modified without corrupting previous contexts. *PopContext* discards the current context and restores the one that was current just before the corresponding *PushContext*.

The function *FindSplitter(e)* returns a propositional variable that is a subformula of e (the choice of formula may be important for efficiency but doesn't matter for correctness). Note that variables which have been asserted (denied) will have been replaced with *true* (*false*) during the *Fold* operation.

The validity checker recursively checks the validity of the formulas obtained by asserting the splitter and folding, and by denying the splitter and folding. If the validity checking function is called with an atomic formula which folds to something other than *true*, a falsifying truth assignment to the atomic propositions can be constructed from the current context and the residual formula. Otherwise, the procedure eventually terminates, having shown that the formula is true under every truth assignment.

3.2 Full logic

The validity checker for the full logic is also described by Figure 1, except that the basic data structures and procedures called are somewhat more complex.

The most significant extension is to the definition of a context. The contexts for the full logic capture properties of equality. Our implementation of these contexts is particularly efficient and will be discussed in more detail below.

As in the propositional case, contexts store assertions about atomic formulas, and a given context may or may not determine

the truth or falsity of an atomic formula. However, in the full logic atomic formulas may be equalities or predicate formulas, where none of the subterms contain *ite* or *write* operations. Whenever the atomic formulas that have been asserted or denied in a context imply or contradict another atomic formula, the data structure is guaranteed to detect and report this fact, except for some omissions noted below.

The functions *FindSplitter* and *IsAtomic* are as described for the propostional case. *Fold* replaces each expression it encounters with the *simplest* equivalent expression in the current context. *Simplest* defines a total order on expressions, and expressions are never simpler than subexpressions. The truth constants *true* and *false* are the least elements, and are simpler than all others. This behavior subsumes the description of *Fold* in the propositional case.

This behavior of *Fold* also ensures that, when $\alpha = \beta$ holds in the current context, expressions $f(\alpha)$ and $f(\beta)$ will be replaced by the same expression (since there is only one *simplest* expression among the expressions equivalent to α and β). The function *Fold* memoizes its results for a given context, so its complexity is linear in the size of the DAG representing its argument.

Expressions containing *read*s and *write*s require a small amount of additional consideration. The validity checker is generally unable to directly prove the equivalence of two stores. Instead, whenever a user wishes to check $\alpha = \beta$, where α and β are expressions yielding store values (e.g. writes), the expressions are transformed into the form $\mathit{read}(\alpha, \mathit{arb\text{-}addr}) = \mathit{read}(\beta, \mathit{arb\text{-}addr})$ where *arb-addr* is a new constant name, distinct from all others in the expression. This formula is valid iff the original is valid, since it asserts that the values of the stores for an arbitrary address must be the same.

Finally, the following transformation is performed automatically by the validity checker:

$$\mathit{read}(\mathit{write}(s, \alpha, v), \beta) \implies \mathit{ite}(\alpha = \beta, v, \mathit{read}(s, \beta)).$$

Otherwise, *read* and *write* can be treated like any other function symbols (in actuality, there are certain heuristics that manipulate *read* and *write*). This transformation is sufficient to eliminate all *write*s during the process of checking.

4 Implementation of contexts

In our logic, expressions can have sub-expressions of arbitrary complexity. We have chosen to implement expressions so that they are *unique*: whenever two expressions are isomorphic, their storage is shared. Therefore, whenever two expressions are syntactically equivalent, their pointers are the same. As in BDD implementations, uniqueness is maintained through a global hash table of all expressions.

4.1 Equality

A context keeps track of equivalences by using the well-known union/find algorithm [Tar75, CLR90], resulting in a data structure that is very fast and space-efficient, both in theory and in practice. The context maintains *equivalence classes* of expressions; two expressions are equivalent if *Find* returns the same value for each expression. Integers are often used to differentiate equivalence classes. Instead, we use one of the expressions in the class which we refer to as the *ECRep* (equivalence class representative).

The $\mathit{Find}(e)$ operation returns the *ECRep* of an equivalence class. The $\mathit{Union}(e_1, e_2)$ function merges the equivalence classes of two expressions. The equivalence of two expressions can be quickly determined with two *Find* operations.

We augment union/find by associating certain contextual information with equivalence classes in fields of the *ECRep*. This information is updated during *Union* operations. The *simplest* expression in an equivalence class, used in the *Fold* operation above, is always in a field of the representative of an equivalence class. There is also a Boolean flag associated with the *ECRep* which is true iff there is a distinct constant in the equivalence class. This facilitates a quick way to recognize an inconsistency when two equivalence classes with distinct constants are merged. We implement *true* and *false* as distinct constants.

The current validity checker is not a complete decision procedure for the logic, because we do not provide full *congruence closure* in the contexts. Congruence closure deals with the interactions of functions with equality [NO80]. For example, the validity checker fails to prove that $f^3(x) = x$ and $f^5(x) = x$ imply the equality $f(x) = x$. However, the validity checker *is* sound — it cannot report that a formula is valid when it is not (a false positive). The omission of congruence closure was intentional, as it has not been necessary in our proofs and is computationally expensive.

4.2 Disequalities

When $\alpha = \beta$ is denied, the result is a *disequality*[1]. Disequalities are more difficult to handle than equalities. Our implementation makes use of a *disequality table*, which is a hash table used to store unordered pairs of expressions. At all times, it is known that $e_1 \neq e_2$ iff $\langle \mathit{Find}(e_1), \mathit{Find}(e_2) \rangle$ appears in the disequality table. Hence, the disequality of two expressions can be checked very rapidly, in the time required for two *Find* operations and a hash-table lookup.

The most costly computation is updating the disequality table during a *Union* operation. Suppose $\mathit{Union}(e_1, e_2)$ modifies the equivalence class representative for e_1. Then, for every disequality in the table of the form $\langle \mathit{Find}(e_1), x \rangle$, a new pair $\langle \mathit{Find}(e_2), x \rangle$, must be entered into the table. To accelerate this operation, all disequalities involving an expression e are stored on a list pointed to by e (so that the *ECrep*s point to lists of all the disequalities referencing them). This list must also be updated on each *Union* operation.

An assertion of disequality is inconsistent with the current context iff it results in an attempt to enter $\langle e, e \rangle$ in the disequality table, for some expression e.

4.3 Contexts and backtracking

The validity checking algorithm requires the ability to assert a formula in a context, then "undo" the assertion so that it can then be denied. Nelson implemented this by carefully removing the assertion from the context [Nel81]. We have a different solution to this problem.

Our solution maintains a global stack of context records. Per-expression contextual information, including the *ECRep*, fields holding the simplest expression and distinct bit, and the list of disequalities involving the expression, are isolated into a distinct record which we call an *ACInfo* (for "assumption context information"). Each *ACInfo* has a pointer back to its context record, and each context record has an *ACInfoChain*, a list of all the *ACInfo*s associated with it.

When information in the *ACInfo* is to be changed, it is first checked whether the *ACInfo* points back to the current context. If not, a copy of the *ACInfo* is made whose context pointer points to the current context record. A pointer to the previous *ACInfo* is stored in a field of the new one. Figure 2 illustrates the data structure. *PopContext* iterates over the *ACInfoChain* of the context record being popped, discarding the current *ACInfo*s and restoring the previous ones.

Each context record also has a list of all the disequality table entries that were defined in the context. *PopContext* removes all of the entries on this list from the disequality table. *PushContext* creates a fresh context, preserving all previous context information.

5 Heuristics

Because validity checking by case splitting is exponential, heuristics are essential for working on large problems. The usefulness of these heuristics varies with the example and the way the expressions are constructed in the symbolic simulator.

[1] as opposed to an *inequality*, such as $\alpha < \beta$.

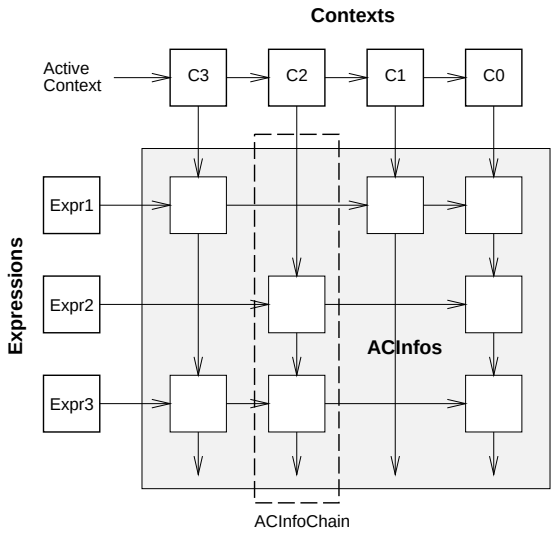

Figure 2: Context Implementation. A context is the information contained in an *ACInfoChain*, *C0* is the base context. Each *ACInfo* contains back pointers to it's expression and context (not shown for clarity).

The order in which splitting expressions are chosen greatly influences the number of steps required by the algorithm, similar to the way that BDDs are sensitive to variable ordering. The most effective splitter selection strategy we have discovered thus far is to search for splitters in large subexpressions first. This heuristic approximately doubles the performance of the validity checker for most of our examples.

As shown in Section 3.1, certain transformations are part of the framework primitives. Other transformations are more complex, and can be selectively enabled.

ITE transformations

Certain *ite* forms contain redundant information. Removing the redundancies when the formula is created is more efficient than removing them during calls to the validity checker. For example, consider the following two transformations:

$$ite\,(\alpha, \alpha, \beta) \implies ite\,(\alpha, true, \beta)$$
$$ite\,(\alpha, \beta, ite\,(\alpha, \gamma, \delta)) \implies ite\,(\alpha, \beta, \delta).$$

Another class of *ite* transformations which we have found to be useful involves recognizing a *not* in the if-part, and transforming the *ite* to remove the *not*:

$$ite(not(\alpha), \beta, \gamma) \implies ite(\alpha, \gamma, \beta).$$

These simple *ite* transformations result in incremental efficiency gains; 20% is typical for our examples.

One idea we should borrow from BDD implementations, but have not yet implemented, is the use of "typed pointers," which have a bit associated with them that changes the interpretation of the expression pointed-to from positive to negative.

If lifting

If-lifting above equalities "lifts" the *ite* if-part(s) out of an equality, moving the equality inside of the resulting *ite*:

$$(ite\,(\alpha, \beta, \gamma) = ite\,(\alpha, \delta, \varepsilon)) \implies ite\,(\alpha, (\beta = \delta), (\gamma = \varepsilon))$$
$$(ite(\alpha, \beta, \gamma) = @\delta) \implies ite(\alpha, (\beta = @\delta), (\gamma = @\delta)).$$

The first transformation demonstrates if-lifting when both arguments of the equality are *ite*s and have identical if-parts α. The second transformation is performed when only one side of the equality is an *ite* and $@\delta$ is a distinct constant (recall that @ distinguishes distinct constants). This transformation pushes the literals down to the leaves of *ite* trees, where they may possibly cancel other literals already there:

$$(@a = ite\,(x, @a, @b)) \implies$$
$$ite\,(x, (@a = @a), (@a = @b)) \implies$$
$$ite(x, true, false) \implies x.$$

When enabled, this transformation occurs a significant number of times. Its effect on our examples is variable, affecting performance by $\pm$40%. If-lifting must be performed with care, as it may destroy sharing of subexpressions. We have discovered that if-lifting in the general case (two *ite*s with if-parts α_1 and α_2) results in an undesirable blow-up in the size of the resultant expressions.

Read and write transformations

In 3.2 we described a *read* transformation which is included in the validity checker to make it sufficiently complete. We have implemented other *read* and *write* transformations which improve efficiency. Consider:

$$ite\,((\alpha = \beta), s, write\,(s, \alpha, \delta)) \implies$$
$$write\,(write\,(s, \alpha, \delta), \beta, read\,(s, \beta))$$

Our testing indicates that this transformation results in a more desirable order of splitter selection, which ultimately results in fewer case splits. For our most complex examples, the validity checker does not finish without this transformation enabled. We are currently investigating the effects of several other transformations involving *read*s and *write*s.

6 Experimental results

We have done extensive testing of our validity checker with two major examples: a simple RISC processor as described by Burch and Dill [BD94b], and a processor being designed at Stanford University as part of the FLASH project [K+94]. For each state variable in the specification of each processor, we use the symbolic simulator to construct an appropriate formula which is then used as input to the validity checker (as was done by Burch and Dill).

The RISC processor is a subset of the DLX architecture [HP90]. The subset we have verified has six instruction types: ALU immediate, 3-register ALU, conditional branch, jump (unconditional branch), load and store. Our example has a 5-stage pipeline with a load interlock.

FLASH is a distributed memory multiprocessor system being developed at Stanford University. FLASH includes a custom memory and interconnect controller with a general purpose protocol processor (PP). PP is a MIPS-based, statically scheduled, fully pipelined, dual-issue RISC processor core with separate instruction and data caches and executes protocol code for shared memory and message passing. PP does not support virtual memory or precise exceptions. However, it employs simple branch prediction and load interlocks. Our PP description contains eight instruction classes: ALU immediate, 3-register ALU, branch-on-equal, jump, jump-register, jump-and-link, load, and store. The PP is a more complex processor and its model is significantly more detailed than the DLX model used by Burch and Dill [BD94b].

For DLX, our specification checks three state variables: the register file, data memory, and program counter. For PP, our specification checks five state variables: those in DLX plus a "next" version of the program counter and a taken-branch bit. The extra state is necessary in PP because of more complex branch instruction semantics.

The heuristics speed up the verification of the DLX processor significantly, and make the verification of PP possible. The timing results for the validity checker with various combinations of heuristics enabled are contained in Table 1.

7 Conclusion

There is obviously a great deal of additional work to be done to reduce the computational complexity (in practice) of the validity

State Variable	Heuristics	DAG size	Execution Time (s)
DLX Register File		400	6.5
	1, 2, 3, 4	492	1.0
DLX Data Memory		400	4.1
	1, 2, 3, 4	491	1.8
	1, 2, 3	439	1.2
DLX PC		501	0.8
	1, 2, 3, 4	1767	2.2
	2, 3	518	0.5
PP Register File	1, 2, 3, 4	735	268.1
PP Data Memory	1, 2, 3, 4	619	825.2
PP Next PC	1, 2, 3, 4	391	0.3
PP Next PC	1, 2, 3, 4	610	80.0
PP Taken Branch	1, 2, 3, 4	583	60.3

Table 1: Expression size and execution time for various processor models. Heuristics: (1) splitter selection, (2) *ite-write*, (3) *ite*, (4) if-lifting. DLX results are reported for no heuristics, all heuristics, and best (known) combination of heuristics. PP examples without heuristics ran out of memory (>512 MB). Measurements conducted on a SUN Sparc-20.

checker. The problem is similar to tautology checking for propositional logic, so presumably more sophisticated techniques from that domain can be applied. However, there are probably ways to exploit the special properties of this domain. Since the problem is provably intractable, there is clearly a need for a larger and more representative sample of benchmark problems, which should capture typical practical problems that are encountered.

Another obvious area for further exploration is making the logic more expressive. Previous work has handled Pressburger arithmetic, and the theory of the reals as well as uninterpreted functions and arrays (which we call stores). However, there has been an efficiency cost. As we attempt to apply our validity checker to a wider range of applications, we expect to encounter situations where greater expressiveness would be an advantage. Further exploration into highly efficient implementations of more general validity checkers will be of great interest.

Acknowledgements

We would like to thank Xiao-Wu Su for developing the description of the Protocol Processor.

References

[BD94a] V. Bhagwati and S. Devadas. Automatic verification of pipelined microprocessors. In *31st ACM/IEEE Design Automation Conference*, 1994.

[BD94b] J. R. Burch and D. L. Dill. Automatic verification of microprocessor control. In *Computer Aided Verification. 6th International Conference*, 1994.

[Bea93] D. L. Beatty. *A Methodology for Formal Hardware Verification with Application to Microprocessors*. PhD thesis, School of Computer Science, Carnegie Mellon University, August 1993.

[C+ 94] F. Corella et al. Multiway decision graphs for automated hardware verification. Unpublished manuscript, August 1994.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[Cyr93] D. Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, December 1993.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

[K+ 94] J. Kuskin et al. The Stanford FLASH multiprocessor. In *International Symposium on Computer Architecture (ISCA)*, 1994.

[LCDM89] P. Lammens, L. Claesen, and H. De Man. Correctness verification of VLSI modules supported by a very efficient boolean prover. In *Proceedings: IEEE International Conference on Computer Design*, October 1989.

[Nel81] G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox PARC, June 1981.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[NO80] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, April 1980.

[SB90] M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, September 1990.

[Sho79] R. E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, April 1979.

[SM95] M. Srivas and S. P. Miller. Applying formal verification to a commercial microprocessor. In *Computer Hardware Description Languages*, August 1995.

[Tar75] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[Wil95] R. Wilson. Verification feels strain. *Electronic Engineering Times*, (840):18–22, March 1995.

[Win95] P. J. Windley. Formal modeling and verification of microprocessors. *IEEE Transactions on Computers*, 44(1):54–72, January 1995.