Automatic Verification of Pipelined Microprocessors

by

Vishal Lalit Bhagwati

Bachelor of Science, University of California at Berkeley (1992)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Science

at the

Massachusetts Institute of Technology

February, 1994

© Massachusetts Institute of Technology, 1994

Signature of Author ______________________________

Department of Electrical Engineering and Computer Science

December 9, 1993

Certified by ______________________________

Srinivas Devadas

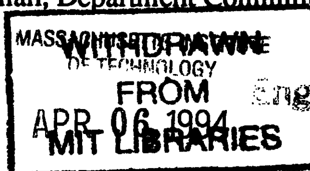
Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor

Accepted by ______________________________

Frederic R. Morgenthaler

Chairman, Department Committee on Graduate Students



Automatic Verification of Pipelined Microprocessors

by

Vishal Lalit Bhagwati

Submitted to the
Department of Electrical Engineering and Computer Science
on December 9, 1993 in partial fulfillment of the requirements for the
degree of Master of Science.

Abstract

This thesis addresses the problem of automatically verifying large digital designs at the logic level, against high-level specifications. In this thesis, a methodology which allows for the verification of a specific class of synchronous machines, namely pipelined microprocessors, is presented. The specification is the instruction set of the microprocessor with respect to which the correctness property is to be verified. A relation, namely the β-relation, is established between the input/output behavior of the implementation and specification. The relation corresponds to changes in the input/output behavior that result from pipelining, and takes into account data hazards and control transfer instructions that modify pipelined execution. The correctness requirement is that the β-relation hold between the implementation and specification.

In this research symbolic simulation of the specification and implementation is used to verify their functional equivalence. The pipelined and unpipelined microprocessor are characterized as *definite machines* (i.e. a machine in which for some constant k, the output of the machine depends only on the last k inputs) for verification purposes. Only a small number of cycles, rather than exhaustive state transition graph traversal and state enumeration, have to be simulated for each machine to verify whether the implementation is in β-relation with the specification. Experimental results are presented.

Thesis Supervisor: Srinivas Devadas

Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I benefited greatly from the work of Srinivas Devadas, my advisor, and Filip Van Aelten, who did ground-breaking work on using string function relations and symbolic simulation for behavioral verification. Professor Devadas helped me significantly during the frustrating moments of the research, and encouraged me to undertake difficult problems and solve them effectively.

I would like to thank the members of the 8th floor VLSI group, my roommates, past and present, and friends for their extensive support during my work.

This thesis is dedicated to my father, who gave me the inspiration to give the best at whatever I undertake, and whose fond memories will live with me forever. Special thanks to members of my family for their continuing support to achieve my educational goals.

The work described in this thesis was done at the Research Laboratory of Electronics of the Massachusetts Institute of Technology.

Contents

1 Introduction 11

1.1 Context 11
1.2 Previous Work 14
1.3 The Work Described in This Thesis 16

2 String Function Relations 19

2.1 Introduction 19
2.2 Concepts and Notation 20
2.3 The "Don't care times" β-Relation 21

3 Automata Theoretic Verification Procedures 25

3.1 Introduction 25
3.2 Binary Decision Diagrams 26
3.3 Image Computation Using BDD's 28
3.4 Procedures for Verification of Finite State Machines 31

4 Microprocessors as Definite Machines for Verification 33

4.1 Introduction 33
4.2 Definite Machines 33
4.3 Verification Properties of Definite Machines and Microprocessors 34
4.3.1 Verification of Definite Machines 34
4.3.2 Microprocessors as Definite Machines 35
4.3.3 β-relation for Verification of Definite Machines 36
4.3.4 Verification of k-definite machines with variable k 39
4.3.5 Pipelined Microprocessors with Data Hazards 41

5 Verification of Pipelined Microprocessors using Symbolic Simulation 43

5.1 Introduction 43
5.2 Pipelined Microprocessors with fixed k 44
5.3 Pipelined Microprocessors with variable k 46
5.4 Observing Specific Variables for Verification 48

5.5 Verification of Pipelined Microprocessors with Interrupts
and Exceptions 49
5.6 Verification of Microprocessors with Dynamically Scheduled Pipelines 52
5.7 Verification of Superscalar Pipelined Microprocessors 54

6 Experimental Results **57**

6.1 Introduction 57
6.2 VSM, a simple RISC processor 57
6.3 Alpha0, a simplified Alpha™ 62

7 Conclusion and Future Work **67**

Bibliography **71**

Chapter 1

Introduction

1.1 Context

Technological advances in the areas of design and fabrication have made hardware systems extremely large. As faster, physically smaller and higher functionality circuits are designed, in large part due to progress made in VLSI, their complexity continues to grow. This makes the design process long and tedious, and error prone. Computer-Aided Design is aimed at alleviating these two major problems in the design process. The first problem is tackled with synthesis programs that automate certain design steps. The second problem is addressed by verification programs that allow a designer to verify the consistency between an initial specification and a derived implementation.

While much progress has been made in the area of synthesis, verification is still lagging behind. Simulation has traditionally been used to check for correct operation of hardware systems, since it has long become impossible to reason about them informally. However, even this is now proving to be inadequate due to computational demands of the task involved. It is not practically feasible to simulate all possible input patterns to verify a hardware design. Sym-

bolic simulation, where symbolic inputs are applied to cover a wider range of input and state space, is practiced increasingly as well. An alternative to post-design verification is the use of automated synthesis techniques supporting a correct-by-construction design style. Logic synthesis techniques have been fairly successful in automating the gate-level logic design of hardware systems. However, more progress is needed to automate the design process at the higher levels in order to produce designs of the same quality as is achievable today by hand. This leads to the need for *independent* verification procedures, and this need is recognized in industry.

There are compelling reasons for verifying hardware to be correct at the design stage, rather than after commercial production and the marketing stage. A comparatively recent alternative to simulation has been the use of formal verification for determining hardware correctness. Formal verification is like mathematical proof. Just as correctness of a mathematically proven theorem holds regardless of the particular values that it is applied to, correctness of a formally verified hardware design holds regardless of its input values. Thus, consideration of all cases is implicit in a methodology of formal verification. We consider a formal hardware verification problem to consist of *formally establishing that an implementation satisfies a specification.* The term *implementation* refers to the hardware design that is to be verified. This entity can correspond to a design description at any level of hardware abstraction hierarchy. The term *specification* refers to the property with respect to which correctness is to be determined. It can be expressed in a variety of ways - as a behavioral description, an abstracted structural description, a timing requirement etc. The implementation and the specification are regarded as given within the scope of any one problem, and it is required to formally prove the appropriate "satisfaction" relation [Gup92].

The ultimate task in verification is to demonstrate that a designed circuit has a correct behavior. Such a circuit can be very large, which makes it imperative that the verification procedure be efficient. The verification task can be split in subtasks, and can be done hierarchically. For instance, a first subtask may consist of demonstrating that a layout has a certain Boolean functionality (which can again be divided into extracting the transistor schematics

from the layout, and proving that the transistor schematic has a correct Boolean functionality). The remaining task is then to demonstrate that a logic design produces an intended overall behavior.

The intended behavior for a synchronous hardware design is not necessarily a specific input / output mapping. Circuits with different degrees of pipelining, or circuits with different degrees of parallelism, produce different input / output functions, but may all exhibit satisfactory input / output behavior. Behavioral verification addresses the problem of verifying that a circuit design exhibits a satisfactory input / output behavior [FVA92]. What constitutes a "satisfactory input / output behavior" depends on the domain of the application.

This thesis considers independent automatic verification of a class of synchronous processors, *pipelined microprocessors*, against behavioral specification. *Pipelining* is an implementation technique whereby multiple instructions are overlapped in execution. Today, pipelining is the key implementation technique to make fast CPUs [PH90]. The work to be done in an instruction is broken into smaller pieces, each of which takes a fraction of the time needed to complete the entire instruction. Each of these steps is called a *pipe stage*. Thus pipelining takes advantage of *instruction level parallelism* to improve the throughput of the microprocessor. The throughput of the pipeline is determined by how often an instruction exits the pipeline. The pipeline designer's goal is to balance the length of the pipeline stages. If the stages are perfectly balanced, then the time per instruction on the pipelined machine - assuming ideal conditions (i.e. no stalls) - is equal to:

Time per instruction on unpipelined machine / Number of pipe stages.

Under these condition, the speedup from pipelining equals the number of pipe stages. The difficulty in designing pipelined microprocessors arises due to control hazards, data hazards and event handling [PH90]. Complex pipelining techniques such as dynamic scheduling, superscalar pipelining, hardware branch prediction etc. are used to obtain maximum throughput in the microprocessor. The design and synthesis of such pipelined microprocessors are

becoming automated, and computer-aided design tools are required to verify the correctness properties of such microprocessors.

1.2 Previous Work

Extensive work has been done on the verification of specific input / output mappings. Although this does not directly address the *behavioral* verification problem, it does so indirectly by providing procedures that can be used as building blocks in a behavioral verification system. Two classes of procedures have been developed for verifying that a circuit exhibits a specific input / output mapping: fully automatic procedures, and procedures that require user intervention. The second class is commonly denoted with the term "theorem-provers", which refer to methods of proving theorems. Theorem provers are built from a general purpose backbone mechanism, which manipulates symbolic expressions by applying inference rules, searching for a way to derive a desired conclusion from a given premise. In contrast, automatic input / output verification procedures are built from representations and routines that are specifically tailored to the problem at hand. Many of the achievements in theorem proving techniques are being superceded by recent advances in automatic procedures. However, theorem proving techniques, such as in [Hun85], [Coh88] and [Joy88]), do have strengths that can be used to advantage for verifying large circuits, namely the allowance for functional abstraction and proof by induction, but they typically require extensive user interaction.

Procedures for verifying strict input / output equivalence between two Finite State Machines (FSMs) were introduced in [CBM89] [Bry87]. This involved exhaustively traversing the State Transition Graph of the product of the two machines, using implicit state enumeration techniques. These are also known as *symbolic simulation* techniques. In symbolic simulation, the input patterns are allowed to contain Boolean variables in addition to constants (0, 1, and X). Efficient symbolic manipulation techniques for Boolean functions allow multiple input patterns to be simulated in one step, potentially leading to much better results that can be obtained with conventional exhaustive simulation. This makes the hardware verification problem, which is NP-hard in general, more tractable and therefore attractive, in practice.

A way of formalizing behavioral equivalence is through string function relations, introduced by Bronstein [Bro89]. Both the implementation and specification are taken to be synchronous machines which have unique string functions associated with them. These string functions map sequences (strings) of input values into sequences of output values. Behavioral equivalence is modeled as a relation between two string functions. Bronstein defined two relations other than strict input / output equivalence: the α- and β-relations, capturing delay and stuttering due to pipelining respectively. Bronstein used the Boyer-Moore theorem prover for the verification of these relations. This allows for the verification of possibly large designs, but requires sophisticated and extensive user interaction.

Sequential logic verification procedures have been extended to allow for differences in the input/output behavior of the specification and implementation by Van Aelten et al [AAD91] [FVA92]. A relation is established between the input/output behavior of the implementation and specification using string function relations. The relation corresponds to changes in the input/output behavior that frequently result from a behavioral or sequential logic synthesis step. It is then possible to automatically verify whether the implementation satisfies the relation with the specification. The definition of the β-relation is extended to general pipelined processors in [FVA92], and the α-relation is subsumed within the extended β-relation. Symbolic simulation methods based on automata theory are used to verify the β-relation. The circuit examples used in this work are those associated with digital signal processing.

A method for verification of pipelined hardware is described by Bose and Fisher [BK89]. They describe a symbolic simulation method for verifying synchronous pipelined circuits based on Hoare-style verification [Hoa69]. To deal with the conceptual complexity associated with pipelined designs, they suggest the use of an *abstraction function*, originally introduced by Hoare to work with abstract data types [Hoa72]. Given a state of the pipelined machine, the abstraction function maps it to an abstract pipelined state. Behavioral specifications for this abstract state space are given in terms of pre- and post-conditions, expressed in propositional logic. By choosing the same domain for the abstract states as the circuit value domain, they are able to automate both the evaluation of the abstraction function as well as verification of the

behavioral assertions. Their technique is demonstrated on a CMOS implementation of a systolic stack. The actual circuit provides inputs to the "abstraction circuit", and the assertions are verified at the abstract state level by a symbolic simulator.

1.3 The Work Described in This Thesis

The work described in this thesis focuses on independent behavioral verification of a class of synchronous machines, namely *pipelined microprocessors*. In my approach, the implementation to be verified is a pipelined microprocessor, and is described in a high-level language similar to BDS [Seg87]. The specification is the instruction set of the microprocessor with respect to which the correctness property is to be verified. It corresponds to an unpipelined implementation of the same instruction set, and is also described in BDS. Logic implementations of these descriptions can be synthesized using a program similar to BDSYN [Seg87]. The correctness requirement is that a β-relation holds between the implementation and specification. The β-relation relates a circuit, that processes inputs at certain relevant time points, and produces outputs at certain relevant time points only, to a circuit of similar functionality that takes and produces relevant inputs and outputs at all time points.

My strategy of verifying the functionality of the implementation against that of the specification involves implicit state enumeration techniques as described in [CBM89]. The pipelined and unpipelined microprocessor are characterized as *definite machines* (i.e. a machine in which for some constant k, the output of the machine depends only on the last k inputs) for verification purposes. In Chapter 4 it is shown that only a small number of cycles, rather than exhaustive traversal, have to be simulated for each machine to verify correctness using the β-relation. The β-relation can be used to model changes in pipelined execution due to data hazards and control transfer instructions. A derivative of the β-relation, the *dynamic β-relation* is developed to verify complex pipelined structures such as interrupt handling, dynamic scheduling, and superscalar pipelines. This makes our methodology viable for large digital systems with complex pipelines.

The thesis is organized as follows. Chapter 2 contains preliminaries on the theory of string function relations. Chapter 3 describes automata theoretic symbolic simulation methods for synchronous machines. In Chapter 4, microprocessors are characterized as definite machines, and their properties that are essential for verification purposes are stated and proved. Chapter 5 contains the methodology used for verifying the pipelined processor implementation against the unpipelined specification. Verification of advanced pipeline structures, such as interrupt handling, superscalar pipelines, and dynamic scheduling are also described in Chapter 5. In Chapter 6, preliminary experimental results are presented. Chapter 7 contains conclusions of the research and gives directions for future work in this area.

Chapter 2

String Function Relations

2.1 Introduction

There are two fundamental ways of describing deterministic sequential machines with functions. One way is a function taking inputs and present states and producing outputs and next states. The other way is with a function taking a sequence (string) of inputs and producing a sequence of outputs, which describes the behavior of a sequential machine for a given initial condition as described in Bronstein's thesis [Bro89]. Both models have particular merits. The first model is indispensable for implementing a circuit, and is useful for many manipulations in the design process, since it is finite. The second model captures more directly the input / output behavior of a circuit, and is useful for expressing and proving certain properties, but since it is infinite (it is a mapping from arbitrarily long input streams to equally long output streams), it seems to be poorly suited for mechanical manipulations. I use both models in my thesis. The string model is used to define behavioral relations between sequential machines, and to prove certain properties of these relations by hand. The operational level is used for the bottom-level mechanized verification work.

This chapter presents the formal correctness requirement we verify, in the form of a relation between the implementation and the specification. The relation corresponds to changes in the input/output behavior that frequently result from a behavioral or sequential logic synthesis step. Formally, we take both the specification and implementation to be synchronous machines and consider the string functions realized by each of them.

This chapter presents string functions and relations between them. Section 2.2 introduces strings, string functions, and notation, as they were presented in [Bro89]. Section 2.3 presents the "don't care times" relation, also known as the β-relation for synchronous machines. Other primitive relations such as the parallelism relation (γ), the encoding relation (δ), the input don't cares relation (ε), the output don't cares relation (ζ) are described in detail in [FVA92].

2.2 Concepts and Notation

Consider an alphabet Σ of values that can appear at the input and output ports of a synchronous circuit. Strings can be defined as finite concatenations of characters in the alphabet. Formally, the set of strings, Σ^*, is defined as $\cup(\Sigma^i)_{i \in w}$ where w is the set of all natural numbers including 0. We use variables u, v, ... for characters, and x, y, ... for strings. The empty string is denoted by ε. Useful operations on strings are as follows:

- . : Concatenate: $\Sigma^* \times \Sigma \rightarrow \Sigma^*$, concatenate two strings (or a string and a character), the second string to the right of the first string. Sometimes the "." will be omitted.

- $|\ |$: Length: $\Sigma^* \rightarrow w$, a length of the string.

- $\leq$: Prefix: $\Sigma^* \times \Sigma^* \rightarrow \{T, F\}$, prefix relation on strings.

  Defined by: ($x \leq \varepsilon \Leftrightarrow x = \varepsilon$) and ($x \leq y.u \Leftrightarrow x = y.u$ or $x \leq y$).

- L: Last: $\Sigma^* \rightarrow \Sigma$. the last character of the string.

  Defined by: $L(x.u) = u$ (and $L(\varepsilon) = \varepsilon$ for totality).

- P: Past: $\Sigma^* \rightarrow \Sigma^*$, all characters of the string except the last one.

Defined by: $P(x.u) = x$ and ($P(\varepsilon) = \varepsilon$ for totality).

- $\uparrow$: To the power: $\Sigma \times w \rightarrow \Sigma^*$, which takes a character and a number n, and returns n repetitions of the character,

- $\downarrow$: At position: $\Sigma^* \times w \rightarrow \Sigma$, which takes a string and a number n, and extracts the nth character from the string. We will use $x{\downarrow}i..j$ as an abbreviation for the string $(x{\downarrow}i).....(x{\downarrow}j)$.

Synchronous systems are constructed from two kinds of building blocks: combinational blocks, implementing a string function f^*, the string extension of a character function f; and registers, which implement a register function R_a that inserts a character a to the left of an input string and cuts off the right most character ($R_a(x.u) = a.x$ and $R_a(\varepsilon) = \varepsilon$).

It is formally demonstrated in [Bro89] that synchronous systems, composed from these primitives, in which every loop contains a register, have a unique associated string function from Σ^* to Σ^*. Such systems are denoted as S_F, S_G, etc., and the corresponding string functions with F and G. It can be seen that these string functions are length preserving (the output string has the same length as the input string) and prefix preserving (if x is a prefix of y, the image of x under the string function is a prefix of the image of y).

As in [Bro89], Greek letters are used for relations between string functions realizable by synchronous machines. For example, $F\ \rho\ G$ means F is in ρ relation with G. The "don't care times" β-relation is of interest to us, and is discussed in Section 2.3.

The alphabets of interest to us contain vectors of Booleans of some length. 0 is used both for the scalar 0 and for vectors containing all 0's. *zero* is the string function taking arbitrary strings x and returning $0\uparrow|x|$. Likewise for 1 and *one*.

2.3 The "Don't care times" β-Relation

Consider an implementation and a specification, both of them being synchronous systems, S_F and S_G respectively, with corresponding string functions F and G. An implementation is considered correct with respect to a specification, if a certain relation holds between the string

functions of the two machines. The relation verified in this thesis is similar to the "don't care times" relation, β, which is defined in [AAD91]. This section presents a formal definition of the β-relation.

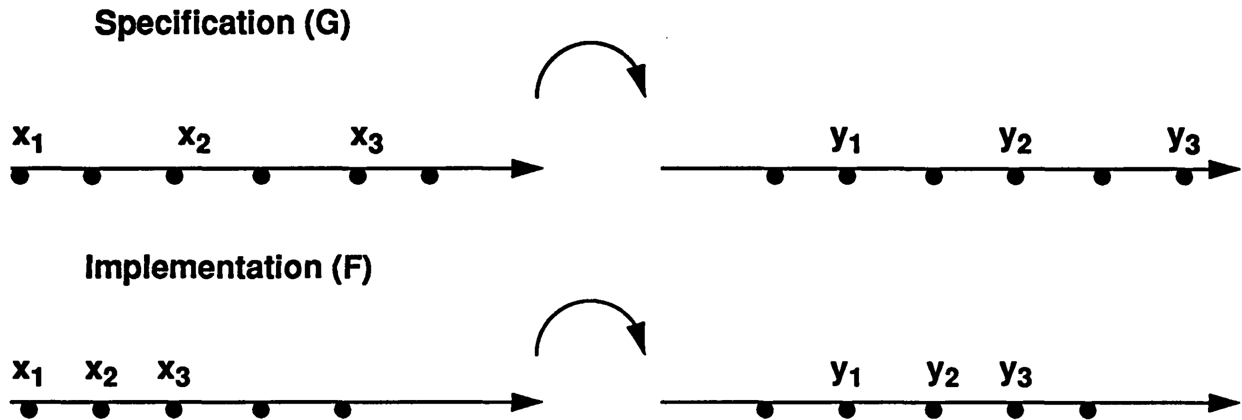


FIGURE 1. The β-relation between *F* and *G*

The "don't care times" relation relates implementation / specification pairs such as the one illustrated in Figure 1. The specification ignores input values at certain time points, and produces irrelevant outputs at certain time points. In addition, the output stream of the implementation is delayed with respect to the one of the specification.

To define the β-relation, a function *Relevant*, which selects the relevant values in a string, i.e. omits the ones at "don't care times", has to be defined first.

Definition 2.3.1 (Relevant): $\Sigma^* \times B^* \rightarrow \Sigma^*$:

$$\text{Relevant}(\varepsilon,\varepsilon) = \varepsilon$$

$$\text{Relevant}(x.u,y.v) = \text{Relevant}(x,y) \quad ,v = 0$$

$$= \text{Relevant}(x,y).u \quad ,v = 1$$

The symbol $\times$ represents the string Cartesian product, which combines strings of equal length. The *Relevant*-function takes a string over an arbitrary alphabet and a Boolean valued string, and returns what remains of the first string after deleting all values for which there is a 0 in the corresponding place in the Boolean valued string.

Let G be the specification and F be an implementation. Let H be a function that filters out the relevant outputs from F and G for verification. Let n be the delay between the output streams of F and G. Then the β-relation is defined as follows.

Definition 2.3.2 (β-relation):

Let H be a length and prefix preserving string function from Σ^* to B^*, realizable by some synchronous machine.

$$F \, \beta_{H,n} \, G \Leftrightarrow \forall x \in \Sigma^* \; s.t. \; |x| \geq n,$$

$$Relevant\,(F(x), R_0\!\uparrow_n o\, H(x)) = G\,(Relevant\,(x\!\downarrow\! 1 \ldots (|x| - n), H(x\!\downarrow\! 1 \ldots (|x| - n)))).$$

This definition says that string functions F and G are in β-relation if applying F to an input stream x, and then picking out the relevant output values, gives the same result as applying G to the relevant input values only. Figure 1 gives an example of applying the β-relation. Here H is a modulo-2 counter and n=1.

Because of the delay of the output stream of the implementation, the filtering function H in the left-hand side of the identity has to be delayed over n cycles. In the right-hand side, the last n characters of the input string have to be eliminated to make the left-hand side and the right-hand side strings of equal length.

Design examples where don't care times occur are:

- A pipelined CPU which has delay slots when taking a branch. Because of the pipelining, not enough information is available for timely execution of a conditional branch. By default the branch is not taken, and when it later appears that the branch should have been taken, the pipeline has to be purged during a number of clock cycles. The outputs produced at those

clock cycles are irrelevant. Moreover, for verification of a pipelined CPU versus an unpipelined CPU, the outputs produced by the unpipelined CPU at certain time points may be irrelevant when comparing outputs with the pipelined CPU at those time points.

- Implementation / Specification pairs as shown in Figure 2. The controller of the implementation is not shown. It repetitively sequences through six states, performing all of the operations of the specification serially. (Taking an input is also considered as an operation.)

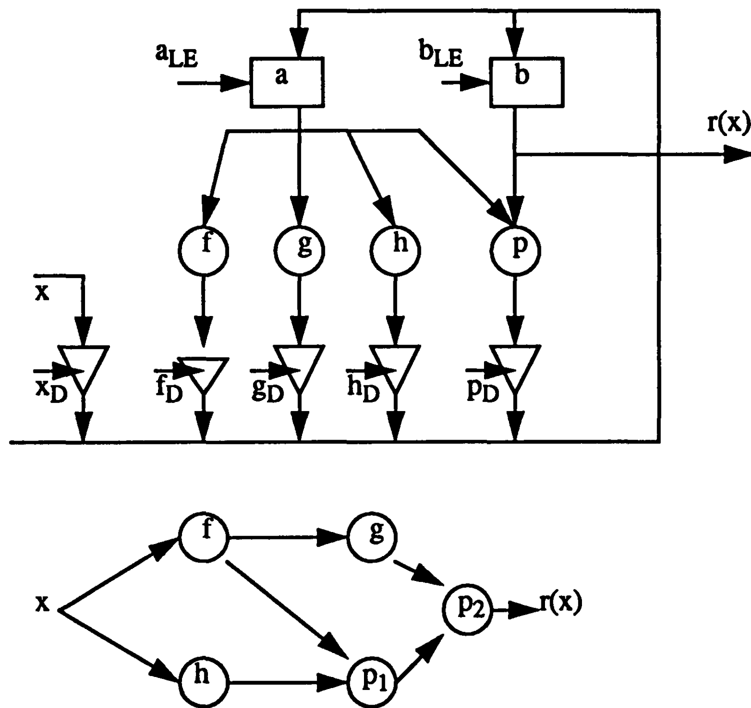


FIGURE 2.Example of an implementation and specification which are in β-relation

The "don't care times" relation is inspired by the stutter-relation in [Bro89], but is slightly different. The α-relation was defined in [Bro89] in the following way:

$$F\,\alpha_z\,G \Leftrightarrow \forall z' \mid |z| = |z'|, \forall x \in \Sigma^*, F(x.z') = z.G(x)$$

It is almost subsumed by the β-relation. If only the delay of F with respect to G is important, and not the characters in z, then $F\ \beta_{one,|z|}\ G$ is equivalent to $F\ \alpha_{|z|}\ G$.

Chapter 3

Automata Theoretic Verification Procedures

3.1 Introduction

This chapter presents an elementary procedure for verification of finite state machines, based on the finite-automaton model. It verifies the input / output equivalence of two deterministic finite state machines. This procedure entails a traversal of the state transition graphs of each machine.

For equivalence checking, a product machine is constructed from the two machines, which produces an output of 1 if and only if the two machines agree on a given input. The transition graph of the product machine has to be traversed to see if all states under all input combinations is produce an output of 1.

There are a number of strategies for traversing a state transition graph. Starting from a certain state, the input combinations leading to a transition out of the state can be enumerated explicitly (i.e. one by one) or implicitly (i.e. in sets). Likewise, the states in the transition graph can be enumerated explicitly and implicitly. For implicit enumeration, different repre-

sentations can be used for sets of input combinations or sets of states (e.g. cubes, sum-of-products, binary decision diagrams). Finally, the traversal can be done depth-first or breadth-first. The methodology that is developed in this work does not commit to any one approach, but the experiments are done with implicit input and state enumeration, based on BDDs, with breadth-first traversal. This is currently the most robust and applicable technique.

Section 3.2 presents the binary decision diagram (BDD)-representation for Boolean functions. Section 3.3 presents image computation routines, for computing the set of states reachable in one step from a given set of states under certain input conditions. These procedures perform implicit input and state enumeration, using BDD's, and can be used for breadth-first traversal. In Section 3.4, a FSM verification procedure is introduced which is used for the methodology developed in this thesis.

3.2 Binary Decision Diagrams

A *binary decision diagram* (BDD) for a function $f(x_1, x_2, x_3, \ldots x_n)$ is a graph with non-terminal vertices annotated with input variables, and terminal vertices annotated with a constant 0 or 1. Figure 3 gives an example of a BDD. All non-terminal vertices have one or more parent vertices and two children vertices, except for the unique root vertex, which has no parents. The terminal vertices have one or more parents and no children. The two edges going down from a non-terminal vertex to their children are annotated with a 0 and a 1. For a given assignment to the input variables x_i, one can go down the decision diagram choosing 1-edges where a variable has a value 1, and 0-edges where a variable has the value 0. The value of the terminal vertex is the value of f under the input combination.

An *ordered* BDD is one where the ordering of the variables is the same along any path from the top vertex to a terminal vertex. A *reduced* BDD has no vertex which has a 1- and 0-edges pointing to the same vertex, or pointing to isomorphic subgraphs. The example BDD in Figure 3 is both ordered and reduced. Reduced Ordered BDDs (ROBDDs) were introduced in [Bry86]. They have a number of properties that make them very well suited for verification. Foremost is the fact that they are canonical, so that equivalence checking between two func-

tions is done by checking for isomorphism between the corresponding BDD's which requires linear time.

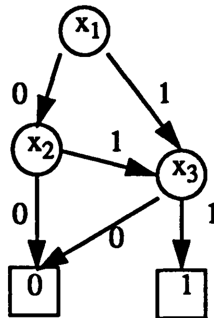


FIGURE 3.Reduced ordered binary decision diagram representing $f = x_1x_3 + \bar{x}_1x_2x_3$

It is not necessary for a Boolean function representation to be canonical to be suitable for equivalence checking. An alternative way of checking equivalence is to apply the XOR-function to the outputs of both functions, and to check for satisfiability of the resulting function. Both operations can be efficiently performed with BDDs. Satisfiability checking requires constant time, and applying a Boolean function to two BDDs requires time proportional to the product of the sizes of the BDDs to which the function is applied.

The apply-operation is also used to construct the BDD for a given implementation of a logic function. Starting from the primary input variables (for which the BDDs are trivial), Boolean functions AND, OR, XOR, XNOR, etc. are applied to BDDs of subfunctions, until a BDD is obtained for the function corresponding to the implementation's output.

The apply-operation can be performed recursively as follows. To compute f_1<*op*> f_2, given ROBDDs f_1 and f_2 with top vertices v_1 and v_2 respectively, do the following:

1. If v_1 and v_2 are terminal vertices, generate a terminal vertex annotated with value (v_1) <*op*> value (v_2).

2. If v_1 and v_2 are annotated with the same variable, construct a vertex annotated with the variable, having as 0-child the result of applying <*op*> to the 0-children of v_1 and v_2, and as 1-child the result of applying <*op*> to the 1-children of v_1 and v_2.

3. If the variables are different, or if one vertex is a terminal vertex, construct a vertex annotated with the variable coming earliest in the variable ordering (i.e. the one that has to appear first along any path from top to terminal vertex). Say that this variable corresponds with v_1. Give to the constructed vertex as 0-child the result of applying <*op*> to v_2 and the 0-child of v_1, and as 1-child the result of applying <*op*> to v_2 and the 1-child of v_1.

The size of the ROBDD is critically dependent on the ordering of input variables. To obtain small BDDs, the variables have to be ordered in such a way that assigning values to the first m input variables ($0 \le m \le n$) can, for the purpose of computing f, be "remembered" with less information than the detailed assignment (which can take 2^m different values). For example, for computing the sum of two integers represented as bit vectors, it is advantageous to interleave the two input vectors, and order the bits from lowest order to highest order. In that case, the only information to be "remembered" from an assignment to the lower order input bits, for the computation of the higher order bits, is one carry-bit.

Such variable orderings are not always feasible. For instance, it has been shown that an ROBDD for a multiplier takes $\Omega(1.09^n)$ space regardless of the variable ordering [Bry91]. Most other practical Boolean functions, however, can be represented efficiently as an ROBDD.

3.3 Image Computation Using BDD's

Image computations constitute the core of the verification procedures to be discussed in Section 3.4. The following is the problem of image computation:

Let f be a function from B^n to B^m, where $B = \{0,1\}$. Let A be a subset of B^n. Compute $f(A)$, the image of A under f, defined as $\{y \in B^m \mid \exists x \in B^n \text{ s.t. } y = f(x)\}$.

An elegant method for image computation is the transition relation method, introduced in [CBM89] and is used for the experiments in this thesis. It works with a BDD representation of the transition relation R corresponding to the function f. The transition relation maps inputs from B^{n+m} into B. It produces an output of 1 if and only if the vector y composed of the last m inputs and the vector x composed of the first n inputs are such that $y = f(x)$.

Subsets A of B^n are in one-to-one correspondence with functions from B^n to B. The characteristic function of a set A is a function that returns 1 if and only if the input is an element of A. The symbol A is used for characteristic functions as well as for sets. Likewise, $f(A)$ is used for characteristic functions and sets.

The image of $f(A)$ consists of y-vectors such that there is an x-vector for which both R and A are 1. The existential quantification can be computed with the smoothing operator, which is defined as follows.

Definition 3.3.1 (Smoothing Operator)

Let f: $B^n \rightarrow B$ be a Boolean function, and $x = (x_{i1}, \ldots, x_{ik})$ a set of input variables of f. The smoothing of f by x is defined as:

$$S_x f = S_{xi1} f \ldots S_{xik} f$$

$$S_{xij} f = f_{xij} + f_{\overline{xij}}$$

In this definition f_a designates the cofactor of f by literal a. For instance f_{xij} is f with x_{ij} restricted to be 1, and $f_{\overline{xij}}$ is f with x_{ij} restricted to be 0. It can be seen that

$$(\exists x \mid R(x,y) \wedge A(x)) \Leftrightarrow S_x(R(x,y) \cdot A(x))$$

and therefore

$$f(A)(x) = S_x(R(x,y) \cdot A(x))$$

Cofactoring with respect to a literal is a trivial operation on BDDs. For instance, cofactoring by x_i is done by deleting the x_i vertices and attaching their 1-children to their parents,

reducing the resulting BDD if necessary. Applying OR- and AND- operations can be done as explained in Section 3.2.

Given the transition relation $\Delta(pi_t, ps_t, ns'_t)$ and a set of states $C_i(ps)$, the next set of states $C_{i+1}(ps)$ can be calculated as follows:

$$E_i(ps, ns') = I \cap C_i(ps) \cap \Delta(pi_t, ps_t, ns'_t)$$

$$C'_{i+1}(ns') = S_{ps} E_i$$

$$C_{i+1}(ps) \leftarrow C'_{i+1}(ps) \cup C_i(ps)$$

where pi_t are the set of primary inputs, ps_t are the set of present state lines, and ns'_t are the transition relation inputs, and $\Delta(pi_t, ps_t, ns'_t) = 1$ if state ns_t reached on applying pi_t to ps_t is $ns'_t = ns_t$.

The smoothing and AND-operation can be performed simultaneously [BCMD90]. The resulting procedure has the same recursive and case-structure as the one for the apply operation. The only difference occurs when the variable corresponding to a newly constructed top vertex has to be smoothed away. In that case the OR of the left child and the right child is computed. This operation is recursive in itself.

The transition relation method can also be used for inverse image computations, where outputs y are given, and inputs x have to be found such that $f(x) = y$.

An alternative method for image computations, making use of Boolean function f only, and not the transition relation R, was introduced in [CBM89]. This method can only be used for forward image computations, not for inverse ones. A good exposition of the method, as well as an extensive set of experimental results, can be found in [TSBS90], which reports that the transition relation method was superior for all the test cases considered.

3.4 Procedures for Verification of Finite State Machines

Using the method described above for image computation, one can check the input / output equivalence of two FSM's. First, construct the product machine, consisting of the original machines with corresponding outputs feeding XNOR gates, and the XNOR gates feeding one AND gate. The product machine is such that it produces an output of 1 if and only if the two component machines agree on a given input.

Second, compute the set of reachable states of the product machine by iteratively computing sets C_i until $C_{i+1} = C_i$. Let s_0 be the initial state of the machine, I the set of inputs under which we need to find the set of next states, and f the next state function.

$$C_0 = \{s_0\}$$

$$C_{i+1} = C_i \cup f(C_i \times I)$$

C_i is the set of states that can be reached in i or fewer steps. The final C_n (which is equal to C_{n-1}) constitutes the set of all reachable states.

With g being the output function, the two machines are equivalent if and only if $g(C_n \times I)$ is a tautology.

Chapter 4

Microprocessors as Definite Machines for Verification

4.1 Introduction

In this chapter, the concept of definite machines is introduced as described in [Koh78]. While the behavior of some synchronous machines depends on remote history, the behavior of others depends only on more recent events. The amount of past input and output information needed in order to determine the machine's future is called the *memory span* of the machine.

Section 4.2 introduces the concept of definite machines, for which only a small amount of past input information is needed to determine the machine's output. Section 4.3 describes certain properties of definite machines that are essential for verification purposes, and these properties form the most important theoretical basis for the efficient verification methodology described in this thesis.

4.2 Definite Machines

Definition 4.2.1 (Definite Machine):

A sequential machine M is called a *definite machine* of order μ if μ is the least integer, so that the present state of M can be determined uniquely from the knowledge of the last μ inputs to M.

A definite machine has *finite input memory.* On the other hand, for a nondefinite machine there always exists at least one input sequence of arbitrary length, which does not provide enough information to identify the state of the machine. A definite machine of order μ is often called a μ-definite machine. If a machine is μ-definite, it is also of finite memory of order equal to or smaller than μ.

The knowledge of any μ past input values is always sufficient to completely specify the present state of a μ-definite machine. Therefore any μ-definite machine can be realized as a cascade connection of μ delay elements, which store the last μ input values, and a combinational circuit which generates the specific output. This realization, which is often referred to as the *canonical realization of a definite machine* is shown in Figure 4. A more detailed treatment of definite machines is given in [Koh78].

4.3 Verification Properties of Definite Machines and Microprocessors

4.3.1 Verification of Definite Machines

Theorem 4.3.1.1 Given two μ-definite machines, let π be the number of possible inputs to each μ-definite machine. Then the two μ-definite machines can be verified with π^{μ} sequences of length μ.

Proof:

We know that the present state of a μ-definite machine can be uniquely determined from the last μ inputs. Since the number of possible inputs is π, the number of all possible permutations of inputs sequences of length μ is π^{μ}. These sequences will enumerate all the unique states of each of the μ-definite machines and the corresponding outputs. If the enumerated states and outputs of the two machines are equivalent, then the two machines are functionally equivalent for the set of π^{μ} input sequences.

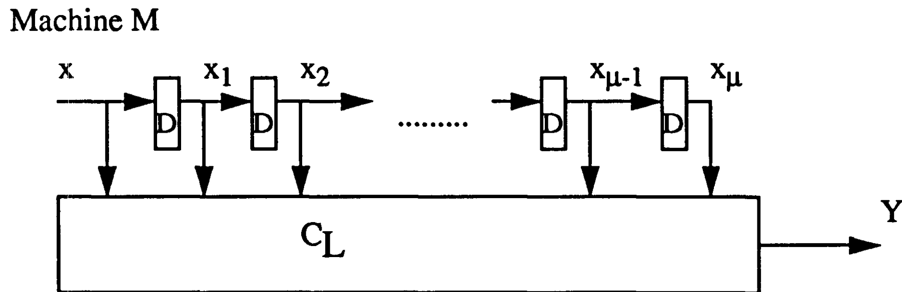


FIGURE 4.Canonical Realization of a μ-definite machine

If the machines are not functionally equivalent even if π^{μ} sequences of length μ produce equivalent present states and outputs of the two machines, then there exists a sequence of length greater than μ which produces a different present state or output, or does not provide enough information to identify the state of either machine. But then it would make the machine nondefinite and with non-finite input memory. This is a contradiction to the original assumption of having μ-definite machines.

Thus the claim holds true, and two μ-definite machines can be verified with π^{μ} sequences of length μ. ❑

4.3.2 Microprocessors as Definite Machines

We argue in this section that a microprocessor, pipelined or unpipelined, can be approximated as a definite machine for verification purposes.

A pipelined microprocessor is designed to have *k* pipeline stages to take advantage of instruction level parallelism to issue a new instruction every cycle. An unpipelined microprocessor consists of the same stages of execution, except that a new instruction is issued only after the previous instruction has completed execution.

Assume *k* is fixed for now, but our argument will hold for variable *k*, e.g pipelined machines which need more information for annulling instructions in delay slots created by control transfer instruction, and for event handling.

Each of the pipelined and unpipelined microprocessors are acyclic machines. An instruction is issued, which performs an operation and changes the state of the machine. This change could include modifying the instruction pointer, register file or memory, all of which are completely observable. The only dependencies are because of register file values. But the register file is completely observable, and differences between the two machine executions can be detected (using the β-relation as will be shown later). Moreover, in each implementation, there are k register stages, each of which feed into combinational logic to produce output. The present state and output of the machine depends only on the previous k inputs and can be determined uniquely, except for register file dependencies.

Thus microprocessors have finite input memory, and can be characterized as definite machines for verification purposes.

4.3.3 β-relation for Verification of Definite Machines

Theorem 4.3.3.1 Two k-definite machines, one an unpipelined machine and the other a pipelined machine can be verified for functional equivalence using the β-relation for synchronous machines.

Proof:

Let S_F be the pipelined k-definite machine, and S_G the unpipelined k-definite machine. Logic transformations are performed on each machine, and string functions are used to filter out the relevant outputs produced by each machine on relevant inputs for verification using the β-relation.

We know that for two k-definite machines with p possible inputs, p^k distinct sequences of length k can verify their equivalence. We need to show that given the same input sequences to the unpipelined and pipelined machines, the same outputs can be obtained but at different times, and the β-relation can be used to verify their equivalence.

For each machine, k is the latency of each instruction. So for both machines, the first k-1 outputs are irrelevant. $n = k-1$ in $\beta_{H,n}$.

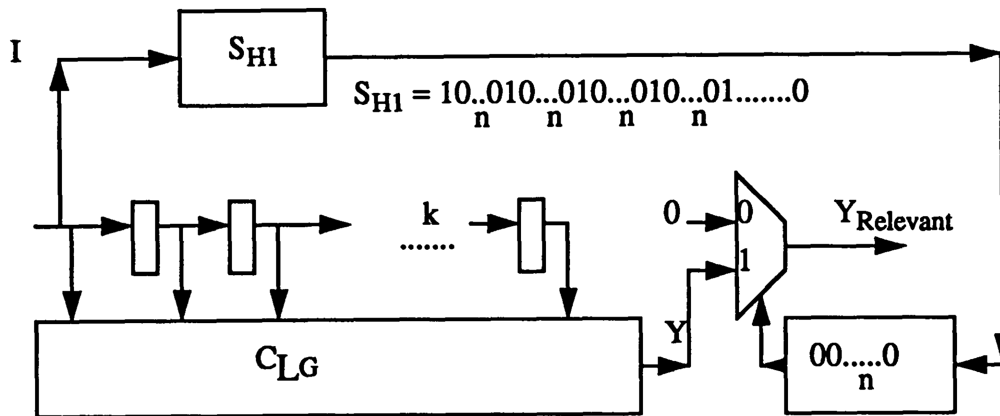


FIGURE 5.Logic Transformation $S_{G'}$ to Unpipelined *k*-definite machine S_G

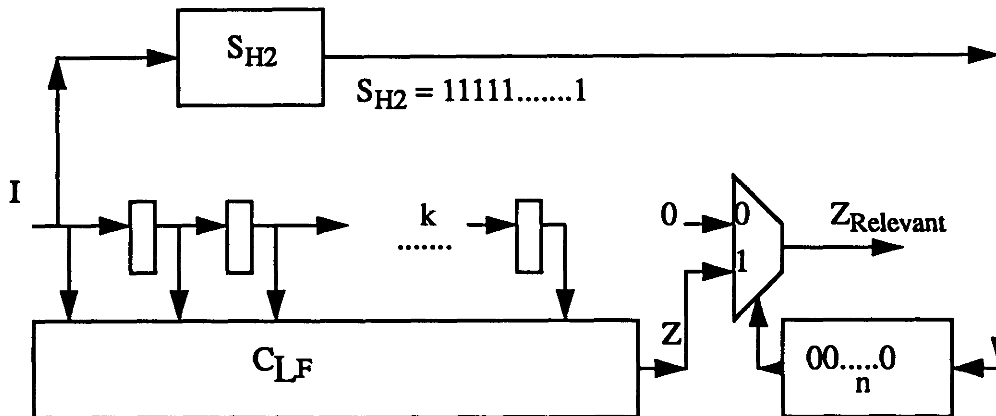


FIGURE 6.Logic Transformation $S_{F'}$ to Pipelined *k*-definite machine S_F

Figure 5 shows the logic transformation $S_{G'}$ and string function S_{H1} to filter the relevant outputs for the unpipelined machine S_G. Figure 6 shows the logic transformation $S_{F'}$ and string function S_{H2} to filter the relevant outputs for S_F (the pipelined machine). For the unpipelined machine, the inputs change every *k* cycles after the previous instruction has completed execution, and outputs are sampled every *k* clock cycles. For the pipelined machine,

inputs change every clock cycle, and outputs are sampled every clock cycle after the first *k-1* cycles. Therefore by construction, comparing the outputs of the logic transformations given p^k input sequences of length *k* verifies the functional equivalence of the two machines.

❑

We have different string functions S_{H1} and S_{H2} for the two machines as shown above. Figure 7 shows a logic transformation $S_{P''}$ for the pipelined machine S_P for which the string functions for both machines are the same, inputs can be changed and outputs can be observed at the same time for both machines.

$S_{H1} = S_{H1'} = 1\underbrace{0..0}_{n}1\underbrace{0...0}_{n}1\underbrace{0...0}_{n}1\underbrace{0...0}_{n}1.......0$

FIGURE 7. Logic Transformation $S_{P''}$ to Pipelined k-definite machine S_P.

The string function S_{H1} to filter out the relevant inputs and outputs also occurs for each memory element i.e. the k registers in the pipelined machine. If a 0 is produced by S_{H1}, the register gets its own value, else it passes its value one register forward. This is a valid logic transformation, and will not alter the state of the machine when S_{H1} produces a 0 as the rest of the logic is purely combinational. Moreover the outputs produced at those times are not relevant. Again, by construction, comparing the outputs of the logic transformations given p^k input sequences of length k verifies the functional equivalence of the two machines.

4.3.4 Verification of *k*-definite machines with variable *k*

In some pipelined machines, k may vary during execution. For example, after control transfer instructions, delay slots are created, and instructions in these delay slots have to be annulled. To effectively annul an instruction, the machine may need information about instructions that may have executed ahead of it. This may increase the order of definiteness of the machine during execution.

Of the k pipeline stages in a pipelined machine, if the target of the current control transfer instruction is known in stage i, $1 \leq i \leq k$, and is effective in the next cycle, all instructions issued and which are currently in stages $1...(i-1)$ have to be annulled. If an instruction in delay slot q, $1 \leq q \leq (i-1)$ modifies the state of the machine (i.e. writes registers, modifies program counter etc., in the case of a microprocessor) in stage j, $q < j \leq k$, then during stage j, instructions $i-q$ beyond stage j have to be known to annul the instruction. So now, the machine becomes $max\ (k, j+i-q)$-definite. In the worst case, we have a $(2k-1)$-definite machine.

Theorem 4.3.4.1 A pipelined k-definite machine, where k varies during execution can be verified against an unpipelined k-definite machine, using the β-relation for synchronous machines.

Proof:

The same logic transformations as shown in Figure 5 and Figure 6 hold for verifying the two machines, except S_{H2} has to be modified not to include the annulled instruction outputs in the relevant values set for the pipelined machine.

Let the control transfer instruction have *m* delay slots. Then S_{H2} is modified as follows:

$S_{H2} = 1\ 1\ 1\ ...\ 1\ 1$ *(as shown in Figure 6)*

except when an instruction is a control transfer instruction, then the next *m* 1's are 0's, i.e. the instructions are annulled and outputs are irrelevant. Any incorrect change in state of the machine, i.e. if any instruction is not annulled, will be detected. So the relevant outputs are filtered out and compared for verifying the functional equivalence of the two machines.

The length of the sequence in the unpipelined machine remains *k*, while in the pipelined machine, it is $max(k, j+i-q)$. We may need sequences as long as $2k-1$ where $k-1$ instructions are annulled. If the xth instruction is a control transfer instruction with $i-1$ delay slots, then

1. $\text{InputUnpipelinedMachine}_{1...x} = \text{InputPipelinedMachine}_{1...x}$,
2. $\text{InputUnpipelinedMachine}_{x+1......k} = \text{InputPipelinedMachine}_{x+i.........max(k, j+i-q)}$, and
3. outputs for $\text{InputPipelinedMachine}_{x+1....x+i-1}$ are not relevant.

For instructions for which outputs are relevant, the length of the sequence is *k*. Therefore, the number of possible instruction sequences is p^k, where p is the cardinality of the instruction set. But to fill up the $i-1$ delay slots, there would be p^{i-1} possible instruction sequences of length $i-1$. Therefore the number of possible instruction sequences becomes $x*p^k*p^{i-1} + p^k$, where x is the probability of having at least one control transfer instruction in the original sequences of length *k*.

If *z* is the number of types of control transfer instructions in the instruction set, then

$$x = \left(\frac{z}{p}\right)^k + \left(\frac{z}{p}\right)^{k-1}\left(1-\frac{z}{p}\right) + \left(\frac{z}{p}\right)^{k-2}\left(1-\frac{z}{p}\right)^2 + \ldots + \left(\frac{z}{p}\right)\left(1-\frac{z}{p}\right)^{k-1}$$

❑

4.3.5 Pipelined Microprocessors with Data Hazards

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. This introduces data hazards. Data hazards occur when the order of access to operands is changed by the pipeline versus the normal order encountered by sequentially executing instructions. Consider two instructions i and j, with i occurring before j. The possible data hazards are as follows:

- Read After Write (*RAW*): j tries to read a source before i writes it, so j incorrectly gets the old value.
- Write After Read (*WAR*): j tries to write a destination before it is read by i, so i incorrectly gets the new value. This cannot happen in an in-order issue pipeline.
- Write After Write (*WAW*): j tries to write an operand before it is written by i. The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination. This hazard occurs in pipelines that write in more than one stage, or issue instructions out of order.

A more detailed description of data hazards and ways of eliminating them can be found in [PH90].

RAW hazards are the most common hazards and occur in most pipelined microprocessors. We will consider only these hazards, since we are evaluating static pipelines, which issue instructions in order and in which each stage executes in one cycle, precluding *WAW* and *WAR* hazards.

The problem of *RAW* hazards can be solved with a simple hardware technique called *bypassing* (or *forwarding*), which is described in [PH90]. Bypassing results in feedback from one stage of a pipeline to one or more stages preceding that one. But this does not alter our definite machine model of microprocessors for verification purposes, as shown in the following theorem.

Theorem 4.3.5.1 Pipelined k-definite machines with bypassing can be verified against unpipelined k-definite machines using the β-relation for synchronous machines.

Proof:

Bypass paths provide correct register values to be used by instructions during execution, and thus facilitate correct execution of the microprocessor. Without bypass paths, one would need to stall the instructions till the source operands become available.

Although bypassing results in feedback from one stage of a pipeline to a stage preceding that, the dependencies are again due to the register file values, which are observable and are allowed for verification purposes. Bypass paths just facilitate these register values to be available at the time when the instructions require them as source operands.

Thus our original model of a definite machine for the pipelined microprocessor is preserved, and the technique mentioned in Theorem 4.3.3.1 can be used to verify the two k-definite machines. ❑

Chapter 5

Verification of Pipelined Microprocessors using Symbolic Simulation

5.1 Introduction

In this chapter, the implementation of the methodology for verification of pipelined microprocessors is explained in detail. The methodology is incorporated into *sis* [SSMS92], a combinational and sequential logic synthesis and verification system. The unpipelined specification and the pipelined implementation are specified in a high-level language BDS. These descriptions are then synthesized into sequential logic using BDSYN, a logic synthesis program, to obtain *slif* netlists.

The user has to specify the properties of the machines, which include k to characterize them as definite machines, and d, the number of delay slots after each control transfer instruction in the pipelined machine. The user also specifies simulation information for the two machines, the use of which will be explained in the sequel.

Section 5.2 includes a discussion of verification of pipelined microprocessors with fixed k. In Section 5.3 verification of microprocessors with variable k is described. Section 5.4

describes some of the implementation details of observing particular variables during the symbolic simulation of each machine. The basic methodology of verification of simple pipelined microprocessors is extended to verify more complex machines with interrupts, traps, exceptions and also dynamically scheduled pipelines. Section 5.5 describes verification of microprocessors with interrupts, traps and exceptions. Section 5.6 describes a method to verify microprocessors with dynamically scheduled pipelines. Section 5.7 gives a brief description of a method to verify superscalar pipelined microprocessors.

5.2 Pipelined Microprocessors with fixed *k*

In this section we consider verification of pipelined microprocessors with fixed k. The operations performed by such machines would be simple ALU operations, memory operations without stalls etc. Microprocessors with variable k will be considered in Section 4.2.

The pseudocode of the algorithm for verification of the pipelined implementation with the unpipelined specification is given in Figure 8.

For each machine, the transition relation is computed for symbolic simulation. To simulate k sequences of instructions, we need to simulate the unpipelined machine for k^2 cycles, and the pipelined machine for $2k-1$ cycles.

For each machine, the input specification functions and output filtering functions are computed from the simulation information provided by the user. The input function specifies what should be the instruction input in a given cycle. For now we are simulating instructions which do not alter the order of definiteness for the pipelined machine for correct execution. For the unpipelined machine, instruction i is fetched in cycle $k(i-1)+1$, and is an input in cycle $k(i-1)+2$. At the $(k(i-1)+2)^{th}$ cycle, we cofactor the transition relation outputs with respect to the inputs such that the cofactored relation corresponds to all instructions that do not alter the order of definiteness of the machine, for all i, $1 \leq i \leq k$. For the rest of the cycles, the transition relation is smoothed with respect to the inputs, since in these cycles, the inputs to the machine are irrelevant. For the pipelined machine, instruction i is fetched in cycle i, and is an input in

cycle $i+1$. At the $(i+1)^{th}$ cycle, we again cofactor the transition relation outputs as described above. Again, for the rest of the cycles, the transition relation is smoothed with respect to the inputs, since in these cycles, the inputs to the machine are irrelevant. The Boolean formula which specifies the inputs for cofactoring is provided by the user.

```
Verify(unpipelinedNetwork, pipelinedNetwork,simulationInfo) {
 Compute input specification function for pipelinedNetwork;
 Compute output filtering function for pipelinedNetwork;
 Compute transition relation for pipelinedNetwork;
 Simulate the pipelinedNetwork for 2k-1 cycles;
   If in a simulation cycle, the output filtering function is 1, sam-
 ple the specified variables and add their formulae to the array var-
 FormulaeP;

 Compute input specification function for pipelinedNetwork;
 Compute output filtering function for pipelinedNetwork;
 Compute transition relation for pipelinedNetwork;
 Simulate the unpipelinedNetwork for k^2 cycles;
   If in a simulation cycle, the output filtering function is 1, sam-
 ple the specified variables and add their formulae to the array var-
 FormulaeU;

 for (i=1, i ≤ K; i++) {
   for (j=1, j ≤ NUM_VARS; j++) {
   verifyBddFormulae(varFormulaeU[i][j], varFormulaeP[i][j]);
   if not equal then the two machines are not functionally
     equivalent and exit;
    }
  }
}
```

FIGURE 8. Algorithm for verifying the *functional equivalence* of the pipelined and unpipelined microprocessors

The output filtering function specifies the cycle in which the variables, which are specified by the user, need to be sampled for verification. For the unpipelined machine, variables are sampled every k cycles, while for the pipelined machine, variables are sampled every cycle after the initial latency of $k-1$ cycles.

Section 5.6 contains a discussion on variables to be observed, and verification of the BDD formulae of these variables.

5.3 Pipelined Microprocessors with variable *k*

We modify our methodology for verification of pipelined microprocessors described in Section 4.1 to include pipelined microprocessors with variable k. The operations that such pipelined microprocessors can perform would be effective annulment of instructions in delay slots of control transfer instructions, event handling, and so on, in addition to those performed by microprocessors with fixed k.

Let d be the number of delay slots for the control transfer instruction. Let instruction I_i be the control transfer instruction, where $1 \leq i \leq k$. The simulation strategy for the pipelined and unpipelined machines is as follows:

- **Pipelined Machine:**

In the pipelined machine, instruction I_i is the input at cycle $i+1$. So we simulate the machine for i cycles and compute the set of next states, as described in Section 4.1. We get the set of reachable states at each cycle and compute the total set of reachable states from the reset state.

At the $(i+1)^{th}$ cycle, we cofactor the transition relation outputs with respect to the inputs that specify that the current set of instructions are control transfer instructions. The Boolean formula which specifies the inputs for cofactoring is provided by the user. We then compute the set of states reachable from the current total set given the new input. In the next d cycles, which are the delay slots, we can compute the next set of reachable states by smoothing away

the inputs from the transition relation, and thus simulate all possible instructions in the delay slots. Thus we get the instruction I_i as specified and only the set of states reachable for that instruction will be accounted for in the new total set of reachable states. The simulation for the cycles that follow is as described in Section 4.1. We simulate the machine for $2*k-1+d$ cycles. This will account for the delay slots in the machine and the verification algorithm will be able to check for proper annulment.

- **Unpipelined Machine:**

In the unpipelined machine, instruction I_i is input at cycle $k(i-1)+2$. So we simulate the machine for $k(i-1)+1$ cycles and compute the set of next states, as described in Section 4.1. We get the set of reachable states at each cycle and compute the total set of reachable states from the reset state.

At the $(k(i-1)+2)^{th}$ cycle, we cofactor the transition relation outputs with respect to the inputs which specify that the current set of instruction are control transfer instructions. We then compute the set of states reachable from the current total set given the new input. The simulation for the cycles that follow is as described in Section 4.1. Thus we get the instruction I_i as specified and only the set of states reachable for that instruction will be accounted for in the new total set of reachable states. We simulate the machine for k^2 cycles.

Certain control transfer instructions, such as conditional branches, sample a value of a status register to decide the next instruction address. Since we are following an implicit simulation strategy, all possible values of the status register are considered, and the next set of states is computed using this information.

The output filtering function for the pipelined machine is modified so as to take into account the delay slots created by the control transfer instruction in the pipeline and the effect of annulment of instructions in these delay slots, i.e. sampling of the variable formulae is not done at the cycles when the instructions in the delay slots would produce outputs. Moreover,

more than one of $I_{1...k}$ can be a control transfer instruction, and accordingly the next states computations can be done and the output filtering function can be specified.

Let z be the number of control transfer instructions, and we simulate one control transfer instruction each time, then the total number of simulations required would be $k*z$. In this scheme, in each simulation, any one of the k instructions is one of the z control transfer instructions. Having more than one instruction as a control transfer instruction in a simulation is not necessary as the particular instruction execution is verified at all of the possible k instruction slots. This improves the efficiency of the methodology, since it does not require simulating all possible combinations of these special instructions.

5.4 Observing Specific Variables for Verification

As mentioned earlier, BDD formulae for variables are to be observed during symbolic simulation of each machine at specific cycles as specified by the output filtering function. The variables to be observed for the two machines are specified by the user. For each microprocessor the variables to be observed may include:

1. General Purpose Registers,

2. Instruction Address Register (the Program Counter PC),

3. Memory Location Contents,

4. Address to Register File and Memory for Read/Write,

5. Instruction Register,

6. ALU Operation.

Once the simulation is completed, the ROBDD formulae of all specified variables at each specified cycle for both machines are obtained. The ROBDD formulae of variables in the pipelined machine at a given cycle are verified with the ROBDD formulae of variables in the unpipelined machine at the corresponding cycle using combinational verification techniques as described in [Bry86]. Given two logic functions, checking their equivalence reduces to a

graph isomorphism check between their ROBDD's G_1 and G_2, and can be done in $|G_1|$ (= $|G_2|$) time.

5.5 Verification of Pipelined Microprocessors with Interrupts and Exceptions

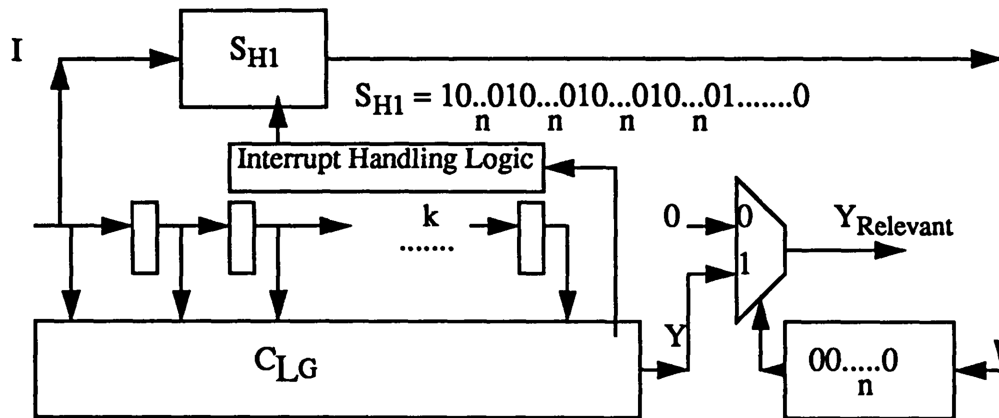


FIGURE 9. Logic Transformation $S_{G'}$ to Unpipelined *k*-definite machine S_G with Event Handling

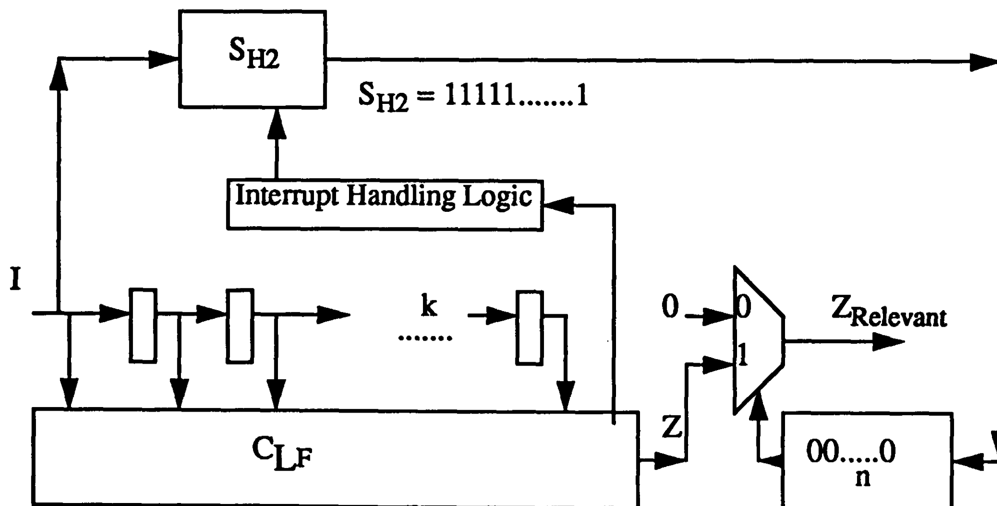


FIGURE 10. Logic Transformation $S_{F'}$ to Pipelined *k*-definite machine S_F with Event Handling

The methodology for verification of simple pipelined microprocessors can be extended to verify the functionality of microprocessors with complex control to handle events such as interrupts and exceptions. These events change the normal flow of instruction execution. Examples of occurrences of these events in the microprocessor are I/O device request, invoking an operating system service from a user program, breakpoint requested by programmer, arithmetic overflow or underflow, page fault, memory-protection violation, hardware malfunctions, using an undefined instruction, misaligned memory access etc. These events can be synchronous or asynchronous, user requested or coerced, user maskable or user nonmaskable, within instructions or between instructions, resumable or terminating. The difficult task of implementing interrupts occurring within instructions where the instruction must be resumed. Another program, often called the interrupt handler, must be invoked to collect the state of the program, correct the cause of the interrupt, and then restore the state of the program before the instruction can be tried again. Even though such events are rare, the hardware must be designed so that the full state of the machine can be saved, including an indication of the event, and the program counter (PC) of the instruction to be executed after the interrupt is serviced. This difficulty is exacerbated by events occurring during the middle of execution, for many instructions also require the hardware to restore the machine to the state just before the event occurred - the beginning of the instruction. If the last requirement is met, then the computer is called *restartable.*

Such events are more difficult to handle in a pipelined machine because the overlapping of instructions makes it more difficult to know whether an instruction can safely change the state of the machine. During the pipelined execution of an instruction, it may change the machine state. Meanwhile, an event can force the machine to abort the instruction's execution before it is completed. The most difficult events are ones that occur within instructions and those that must be restartable.

Figure 10 shows the logic transformation applied to verify a pipelined *k*-definite machine which contains logic to handle events such as interrupts and exceptions. The interrupt handling

logic is designed in such a way so that when an event occurs, it takes the following steps to save the pipeline state safely:

1. Force a trap instruction into the pipeline on the next instruction fetch.

2. Until the trap is taken, turn off all writes for the faulting instruction and for all instructions that follow in the pipeline. This prevents any state changes for instructions that will not be completed before the event is handled.

3. After the event-handling routine receives control, it immediately saves the PC of the faulting instruction. This value will be used to return from the event later. Multiple PC's may be required to save the state of the machine in the case of delayed branches.

If the pipeline can be stopped so that the instructions just before the faulting instructions are completed and those after it can be restarted from scratch, the pipelined is said to have *precise interrupts*. Ideally, the faulting instruction would not have changed state, and correctly handling some interrupts requires that the faulting instruction have no effects. For other events, such as floating-point exceptions, the faulting instruction on some machines writes its result before the event can be handled. In such cases, the hardware must be prepared to retrieve the source operands, even if the destination is identical to one of source operands. Supporting precise interrupts is a requirement in many systems, while in others it is valuable because it simplifies the operating system interface. At a minimum, any machine with demand paging or IEEE arithmetic trap handlers must make its interrupts precise, either in hardware or with some software support. A comprehensive description of events and hardware and software techniques for handling them in pipelined microprocessors is given in [PH90].

The verification of such a pipelined microprocessor against an unpipelined microprocessor, each characterized as a definite machine, can be done in a similar fashion as is done for verification of definite machines with variable k to handle branch delay slot instruction annulment. The event is simulated on each machine in each of the k sequences of instructions.

Theorem 5.5.1 A pipelined *k*-definite machine with interrupts and traps can be verified against an unpipelined *k*-definite machine with interrupts and traps using the β-relation for synchronous machines.

Proof:

Figure 10 shows the logic transformation applied to a pipelined machine with event handling logic for verification against a similar unpipelined machine, whose logic transformation is shown in Figure 9. S_{H1} and S_{H2} are the output filtering functions for the unpipelined and pipelined machine respectively. Let an event occur when instruction j, $1 \le j \le k$, is executing. When an event is detected, the interrupt handling logic modifies S_{H1} and S_{H2} in such a way so that 0's are inserted in positions of the filtering function when the event is being handled, and when the event handling is completed, the output filtering functions are restored as they would be under normal execution. This output filtering function is also called the *dynamic β-relation*, since occurrences of such events are not known before the instruction is executed, and the output filtering function is changed on the fly during execution of instructions on each machine. With such a strategy, the verification algorithm will determine whether the pipelined machine is designed with correct interrupt handling logic.

❑

5.6 Verification of Microprocessors with Dynamically Scheduled Pipelines

So far we have assumed that our pipeline fetches an instruction and issues it. But in some cases, so as to handle multi-cycle instructions, the pipeline may be stalled if there is a data dependence between an instruction in the pipeline and the fetched instruction in the pipeline. The issuing is ceased until the dependence is cleared. This is called *static scheduling*. To gain higher throughput and lower latency for instructions in the pipeline, *dynamic scheduling* can be implemented, whereby the hardware rearranges the instruction execution to reduce stalls. This offers the advantages of handling some cases when dependencies are unknown at compile time, and of simplifying the compiler. It also allows code that was compiled on one pipeline

in mind to run efficiently on another pipeline. These advantages are gained at a significant increase in hardware complexity. *Scoreboarding* and *Tomasulo's algorithm* are two techniques that are used to reduce the cost of data dependencies, especially in deeply pipelined machines. *Hardware branch prediction* is a dynamic technique for handling branches: to predict whether a branch will be taken, and to find the target more quickly. A comprehensive description of incorporating such techniques in hardware is given in [PH90].

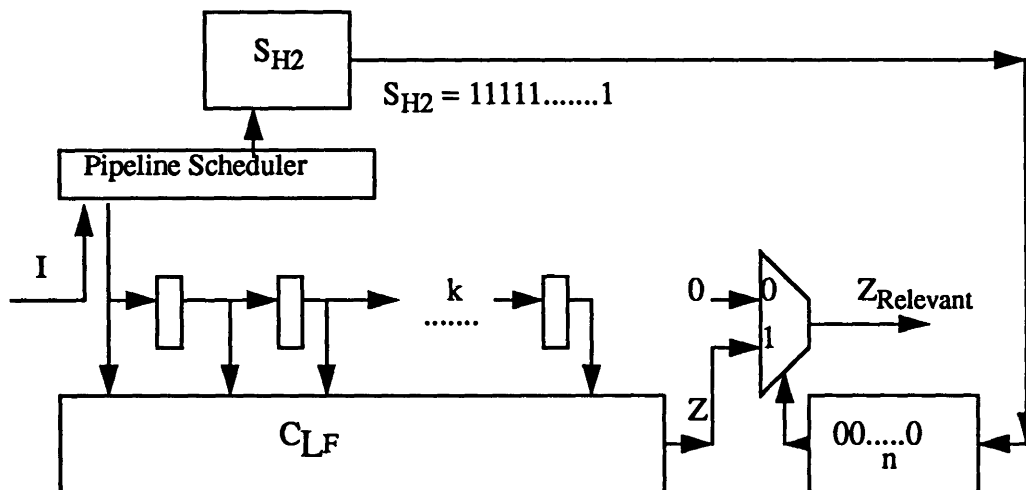


FIGURE 11.Logic Transformation $S_{F'}$ to Dynamically Scheduled Pipelined *k*-definite machine S_F

Figure 11 shows the logic transformation applied to a pipelined machine with a dynamically scheduled pipeline for verification purposes. The verification of pipelined microprocessors with out-of-order execution is a very difficult problem with the methodology proposed in this thesis. In this methodology, the output filtering functions S_{H1} and S_{H2} of the unpipelined and pipelined machines respectively filter out the outputs at appropriate times and the verification of state is done using combinational verification techniques assuming in-order execution of instructions. But in the case of out-of-order execution, the unpipelined machine still exe-

cutes in-order, while the pipelined machine instructions are issued out-of-order to reduce stalls due to data and control hazards.

A modified *dynamic* β*-relation* can be used to verify the pipelined machine with dynamic scheduling against an unpipelined machine. Verification of the state of the two machines during instruction execution would not make sense, since the instructions are executed out-of-order in the pipelined machine. But during execution in the pipelined machine, if the current set of instructions that have been executed are in-order, although they might be out-of-order amongst themselves, the state of the machine at that time point can be verified with the state of the unpipelined machine when this set of instructions have completed execution. For this purpose, the output filtering function or the β-relation, has to be modified during execution as the instructions are being scheduled by the scheduler. Using this strategy, it may be the case that the state of the machines is observed at the very end of the execution of all sets of instructions. When the states of the two machines are compared at appropriate time points as determined by the output filtering function, or the β-relation, the verification algorithm will determine whether the pipelined machine is designed correctly to handle dynamic scheduling of instructions in the pipeline.

5.7 Verification of Superscalar Pipelined Microprocessors

In a superscalar machine, the hardware can issue a small number (usually 2 to 4) of independent instructions in a single clock. However, if the instructions in the instruction stream are dependent or do not meet certain criteria, only the first instruction in the sequence will be issued. To get this high throughput for the machine, multiple functional units are included.

The verification of superscalar pipelined microprocessors can be done using the methods described above. The microprocessor can be characterized as a k-definite machine, except that if q is the order of superscalarity, then instead of k sequences of instructions, we need to issue kq sequences of instructions to verify that the superscalar microprocessor is functionally correct. The pipeline scheduler will schedule at most q instructions in a given cycle. The β-rela-

tion between the two machines has to be modified to successfully verify the functionality of the superscalar machine.

Let m instructions, $1 \le m \le q$, complete executions at a given time. The output filtering function of the unpipelined machine S_{H1} is modified so as to sample the state of the machine after these m instructions have completed execution. This is possible since in-order superscalar execution is assumed. The output filtering function of the pipelined machine S_{H2} samples the outputs when a given instruction completes execution, and determines the value of m. Therefore, here too, we use the dynamic β-relation to verify the correctness of the superscalar pipelined microprocessor.

Chapter 6

Experimental Results

6.1 Introduction

To demonstrate the feasibility of this methodology, experiments were performed on two pipelined microprocessor designs. Section 6.2 describes experiments performed on VSM, a simple microprocessor designed for experimental purposes. Section 6.3 described experiments performed on $Alpha_0$, a simplified Alpha™, which is more representative of architectures that exist in industry today.

Experiments on each design showed that the methodology is very sound, but is limited by the computational power of BDD's. This resulted in a significant condensation of designs, but enough functionality was retained to show that the methodology can be applied to sophisticated pipelined designs.

6.2 VSM, a simple RISC processor

The first design, VSM, is a simple RISC pipelined microprocessor. Table 1 gives a description of the instruction set of VSM. Figure 12 shows a schematic diagram of the pipe-

lined VSM, and Figure 13 shows the schematic diagram of the unpipelined VSM. Some details have been omitted for clarity. The salient features of VSM are as follows:

a. 13-bit single format instructions.

b. 4-stage static pipeline for the pipelined machine.

c. Eight 3-bit general purpose registers.

d. 3-bit ALU operations.

e. One delay-slot after the branch instruction,

f. 5-bit Instruction Address Register (PC).

TABLE 1. VSM Instruction Set

Format	<12:10>	<9>	<8:6>	<5:3>	<2:0>	
	Opcode	L	Ra/Disp	Rb/Lit	Rc	

Instruction	Opcode	Operation
add	000	If L=0, Rc←<Ra>+<Rb>, else Rc←<Ra>+Lit
and	010	If L=0, Rc←<Ra> AND <Rb>, else Rc←<Ra> AND Lit
br	100	Rc←PC, PC←PC+Disp
or	011	If L=0, Rc←<Ra> OR <Rb>, else Rc←<Ra> OR Lit
xor	001	If L=0, Rc←<Ra> XOR <Rb>, else Rc←<Ra> XOR Lit

For the VSM, the order of definiteness of the machine, k equals 4, and the number of delay slots after branch instruction d equals 1. A simulation information file is created for implementing the verification methodology. For the VSM, a sample simulation information file would like as follows:

```
# Simulation Information File for VSM.
r #Simulate a reset cycle
0 #Simulate all instructions except for control transfer
```

```
0
1 #Simulate control transfer instructions
0
```

With the simulation information, inputs for the machine can be determined for selective simulation of particular instructions. The above information says that a reset instruction is to be simulated first, followed by a sequence of two sets of all instructions except for control transfer instructions, followed by a control transfer instruction, and then again a set of instructions except for control transfer. The unpipelined machine is simulated for k^2+r cycles, where r is the number of reset instructions, while the pipelined machine is simulated for $2k-1+r+cd$ cycles, where c is the number of control transfer instructions. The outputs have to be sampled at times as specified by the β-relation between the two machines. The output filtering function for this input specification would be as follows:

UNPIPELINED: `1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1`

PIPELINED: `1 0 0 0 1 1 1 0 1`

where a '1' in filtering function implies that at this time point the specific outputs are sampled, and a '0' in the filtering function indicates that the outputs have to be ignored at that time point.

The variables observed include:

a. The general purpose registers,

b. ALU operations,

c. Instruction Address Register.

To reduce the number of latches, and thus speed up the symbolic simulation, we experimented with having only one general purpose register in the machine, and observed the read/write addresses to the register file to emulate the effect of having all eight registers. This improved the efficiency of our methodology. Simulation time for the unpipelined VSM was *175 sec*, while that for the pipelined VSM was *292 sec* on a Sun SPARCstation 10. Verification of variable formulae was done using ROBDD-based combinational techniques [Bry86].

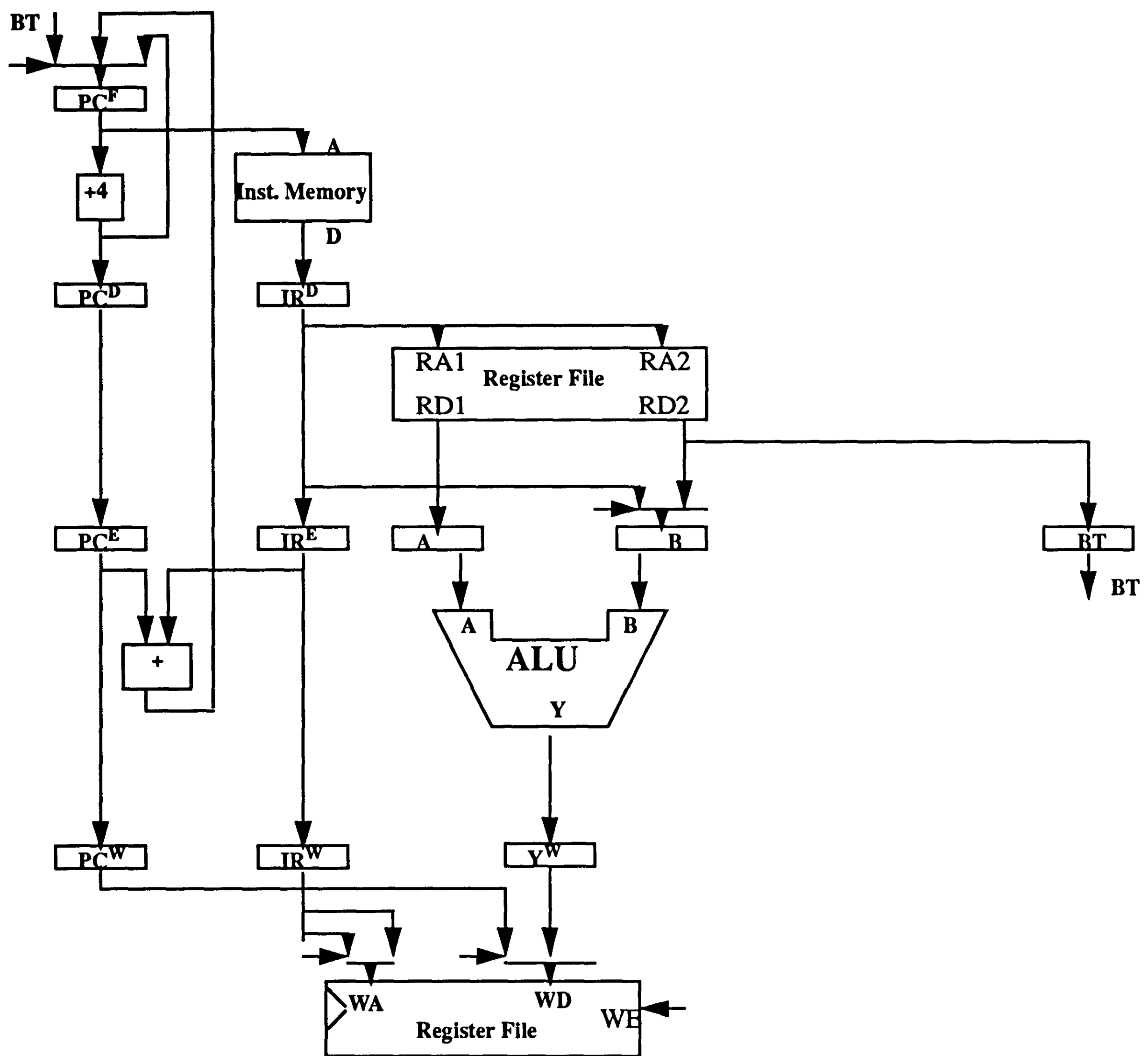


FIGURE 12.The Pipelined VSM

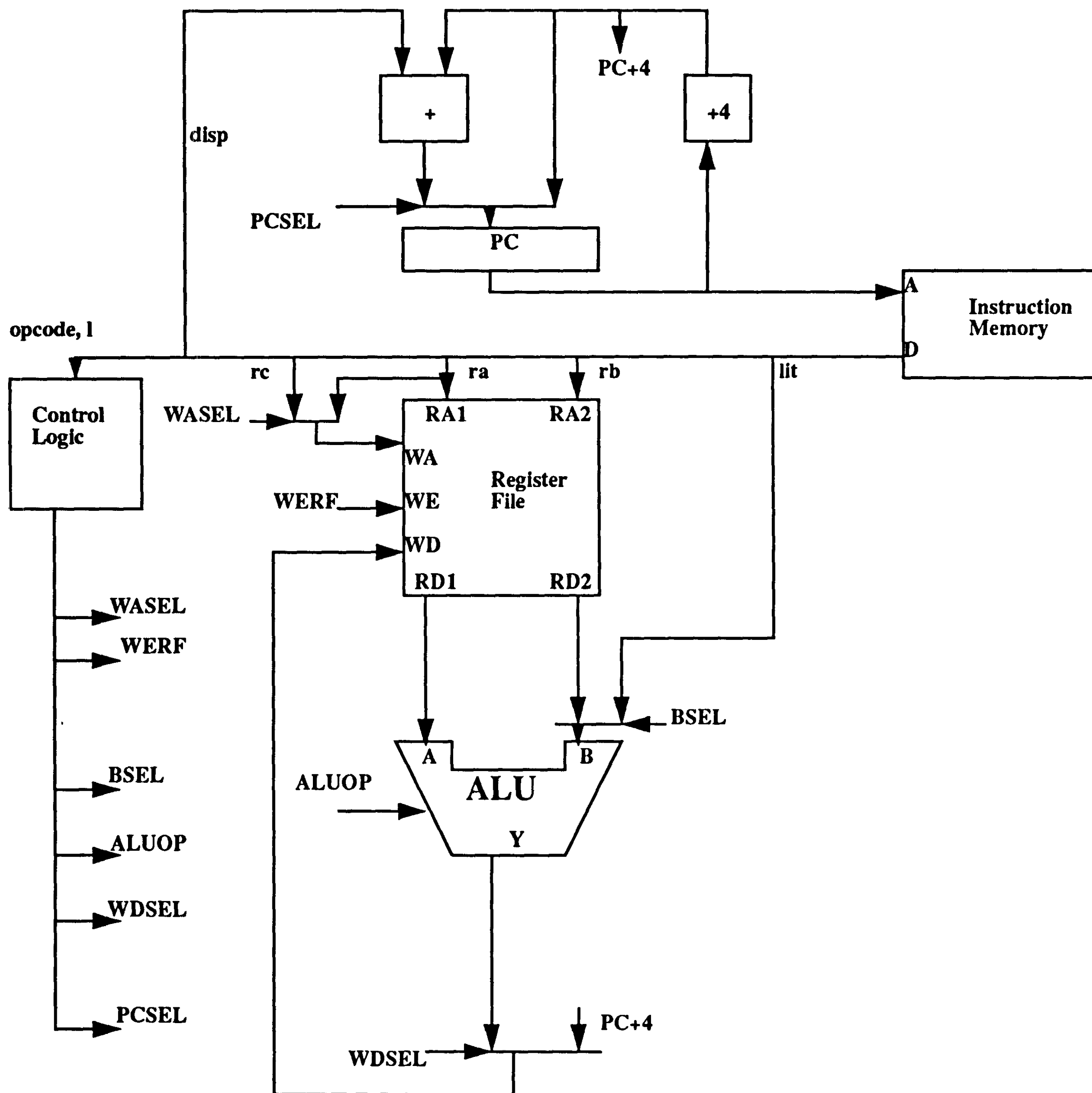


FIGURE 13.The Unpipelined VSM

6.3 Alpha$_0$, a simplified Alpha™

The second design, Alpha$_0$, is a subset of the DEC-Alpha™ microprocessor. Table 2 gives a description of the Alpha$_0$ instruction set. Figure 14 shows a schematic diagram of the pipelined implementation of the Alpha$_0$, while Figure 15 shows a schematic diagram of the unpipelined implementation of the Alpha$_0$. Some details have been omitted for clarity. Specifically, bypass paths are not shown. We initially experimented with a full 32-bit design of the Alpha$_0$, but limitations in computational capabilities of BDD's compelled us to condense the design, the features of which include:

a. Load/Store RISC architecture,

b. 32-bit fixed format instructions,

c. 5-stage static pipeline for the pipelined machine,

d. Thirty-two 4-bit registers,

e. 4-bit ALU operations,

f. One delay-slot after each control transfer instruction,

g. 5-bit Instruction Address Register.

TABLE 2. Alpha$_0$ Instruction Set

Format	<31:26>	<25:21>	<20:16>	<15:13>	<12>	<11:5>	<4:0>
Operate	Opcode	Ra	Rb	000	0	Function	Rc
Op with Literal	Opcode	Ra	Literal		1	Function	Rc
Memory	Opcode	Ra	Rb	disp.m			
Branch	Ra	disp.b					

Instruction	Opcode	Function	Operation
add	0x10	0x20	Rc ← <Ra> + <Rb> | Lit
and	0x11	0x00	Rc ← <Ra> AND <Rb> | Lit
br	0x30	-	Ra ← PC, PC ← <PC> + 4 · SEXT (disp.b)
bf	0x39	-	Update PC, EA←<PC> + 4 · SEXT(disp.b), If <Ra>=0, PC←EA

TABLE 2. **Alpha$_0$ Instruction Set**

bt	0x3D	-	Update PC, EA←<PC> + 4 · SEXT(disp.b), If <Ra> ≠ 0, PC←EA
cmpeq	0x10	0x2D	If <Ra> = <Rb>, then Rc ← 1 else Rc ← 0
cmple	0x10	0x6D	If <Ra> ≤ <Rb>, then Rc ← 1 else Rc ← 0
cmplt	0x10	0x4D	If <Ra> < <Rb>, then Rc ← 1 else Rc ← 0
jmp	0x36	-	Update PC, EA←<Rb>∧ 0xFFFFFFFA, Ra←PC, PC←EA
ld	0x29	-	EA←<Rb> + SEXT(disp.m), Ra←Memory[EA]
or	0x11	0x20	Rc ← <Ra> OR <Rb> | Lit
sll	0x12	0x39	Rc ← <Ra> SLL <Rb>$_{5:0}$ | Lit$_{5:0}$
srl	0x12	0x34	Rc ← <Ra> SRL <Rb>$_{5:0}$ | Lit$_{5:0}$
st	0x2D	-	EA←<Rb> + SEXT(disp.m), Memory[EA]←<Ra>
sub	0x10	0x29	Rc ← <Ra> - <Rb> | Lit
xor	0x11	0x40	Rc ← <Ra> XOR <Rb> | Lit

For the Alpha$_0$, *k* equals 5, and *d* equals1. For the Alpha$_0$, a sample simulation information file would like as follows:

```
# Simulation Information for Alpha0.
r #Simulate a reset cycle
0 #Simulate all instructions except for control transfer
0
1 #Simulate control transfer instructions
0
0
```

The above information says that first a reset instruction is to be simulated, followed by a sequence of two sets of all instructions except for control transfer, followed by a set of control transfer instructions, followed by a sequence of two sets of instructions except for control transfer. The output filtering function for this input specification would be as follows:

<u>UNPIPELINED:</u> 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1

PIPELINED: 1 0 0 0 0 1 1 1 0 1 1

The output filtering functions for each machine can be modified to verify the functionality of each machine for interrupt handling as described in Section 5.5. Out-of-order execution can be verified for the pipelined machine as described in Section 5.6.

The variables observed are the same as for the VSM. We also observed memory read/write addresses. Again, we used the single general purpose register model for the $Alpha_0$ to speed up the symbolic simulation. In order to reduce the complexity of the machine, we simplified the ALU to have only the *and*, *or*, and *cmpeq* operations, and further have 4-bit operations. The ALU operations issued by instructions were observed to emulate a fully functional ALU, while the more complex ALU can be verified using combinational techniques [Bry86]. Simulation time for the unpipelined $Alpha_0$ was *23 min*, while that for the pipelined $Alpha_0$ was *43 min* on a Sun SPARCstation 10. Verification of variable formulae was done using ROBDD-based combinational techniques.

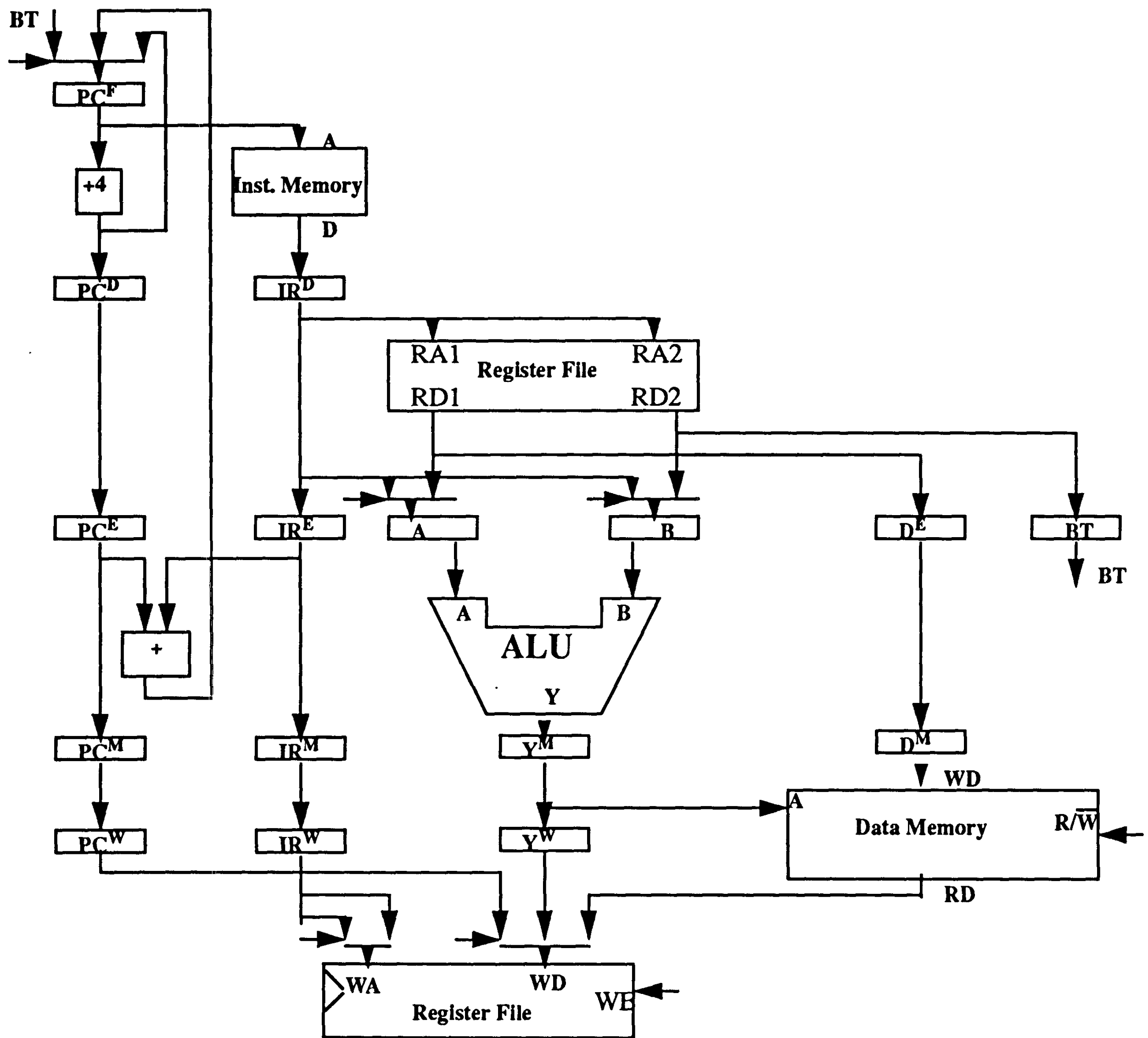


FIGURE 14.The Pipelined $Alpha_0$

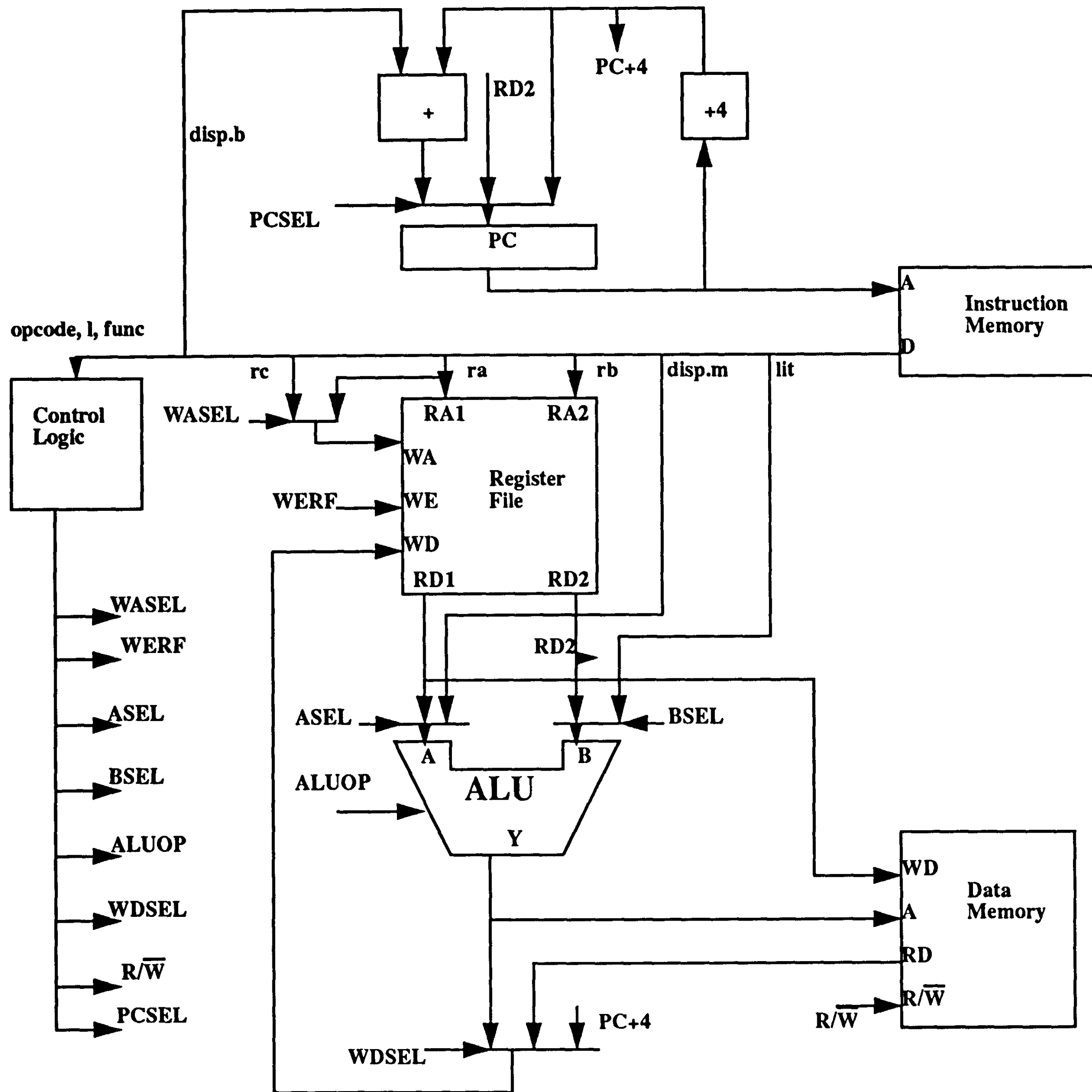


FIGURE 15. The Unpipelined $Alpha_0$

Chapter 7

Conclusion and Future Work

The main contribution of this thesis is that a sound methodology for verification of pipelined processors has been developed. The specification is taken to be an unpipelined implementation of an instruction set, and the implementation is the pipelined microprocessor to be verified. Behavioral equivalence is formalized as a relation between the string functions associated with the implementation and the specification. String functions capture the input / output behavior of a synchronous machine in a pure form, without reference to the internal state of the circuit. Although the implementation is pipelined with respect to the specification, the β-relation bridges this difference. The methodology is based on verifying the defined β-relation using symbolic simulation, and is used for verification of complex pipelined microprocessors.

Recently a number of procedures have been introduced based on BDD representations and implicit enumeration techniques, which efficiently perform certain verification tasks. The primary computation cost in these methods is BDD manipulation, and they cannot be directly applicable to the verification of large industrial designs. The basic procedures can be used indirectly, and sufficient conditions for an overall correctness requirement can be derived. As

was done with the $Alpha_0$, described in Chapter 6, non-essential combinational logic that increases BDD size can be discarded for improved efficiency.

Pipelined microprocessors are characterized as *definite machines*, and verification properties of such machines have been developed in this research. This leads to a formulation that if a microprocessor is a definite machine of order k, then in symbolic simulation, only k sequences of all possible inputs are required to verify its correctness with another k-definite machine. The β-relation for k-definite machines was developed to verify the correctness of a pipelined microprocessor against an unpipelined microprocessor. A *modified* β-*relation* is used to verify the correctness of pipelined microprocessors with control hazards. Methods of verifying correctness of complex pipelined structures such as interrupt handling, dynamic scheduling, and superscalar pipelines have been developed using the *dynamic* β-*relation*, where the string function relating the input / output behavior of the implementation and the specification is modified during execution for filtering out the outputs at appropriate times according to the instruction execution status. This makes the methodology applicable to designs used in industry, at present, and is scalable to industrial designs of the future, because of its flexibility in handling complex pipelined structures.

The strategy described in this thesis is highly automatic and requires very little user interaction. This makes the methodology a valuable tool for all hardware design engineers in industry, at present and in the future. The designer has to provide

a. The descriptions of the specification and implementation,

b. The properties of the machines, such as k, number of delay slots after control transfer instructions, etc.,

c. List of variable formulae to observed and verified to determine correctness,

d. Variable lists to create BDD formulae to cofactor transition relation outputs to simulate specific instructions, i.e. the opcodes of these instructions.

Two designs were verified with the methodology described in this thesis. One design is an experimental RISC microprocessor, VSM. The other design $Alpha_0$, is more representative of

industrial designs existing in industry today, since it is a subset of the DEC-Alpha™. Both of these experiments showed that the methodology can be applied successfully to a number of pipelined microprocessors, with a variety of simple and complex features.

There are several opportunities for further work. First, more work can be done in developing better ordering constraints for constructing BDD's given a logic circuit, so that symbolic simulation can be done efficiently. A second topic of fundamental research would be to develop better representations of the transition relation of sequential machines, to bypass the computational limitations of BDDs.

Bibliography

[FVA92] F. Van Aelten. *Automatic Procedures for the Behavioral Verification of Digital Designs*. Ph.D. Dissertation, MIT, 1992.

[AAD91] F. Van Aelten, S.Y. Liao, J. Allen, and S. Devadas, Automatic Generation and Verification of Sufficient Correctness Properties for Synchronous Processors. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 183-187, November 1992.

[BK89] S. Bose, and A. Fisher. Verifying pipelined hardware using symbolic logic simulation. In *Proceedings of the IEEE Conference on Computer Design*, pages 217-221, 1989.

[Bro89] A. Bronstein. *MLP: String-Functional Semantics and Boyer-Moore Mechanism for the Formal Verification of Synchronous Circuits*. Ph.D. Dissertation, Stanford, STAN-CS-89-1293,1989.

[Bry86] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. In *IEEE Transactions on Computers*, volume C-35, pages 677-691, August 1986.

[Bry87] R. Bryant. A methodology for hardware verification based on logic simulation. Technical Report CMU-CS-87-128, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, June 1987.

[Bry91] R. Bryant. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication. In *IEEE Transactions on Computers*, volume 40, pages 205-213, February 1991.

[BCMD90] J. Burch, E. Clarke, K. McMillan, and D. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proceedings of the 27th Design Automation Conference*, pages 46-51, June 1990.

[Coh88] A. Cohn. A Proof of Correctness of the VIPER Microprocessor: The First Level. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 111-128. Kluwer Academic Publishers, 1988.

[CBM89] O. Coudert, C. Berthet, and J.C. Madre. Verification of Sequential Machines Using Boolean Functional Vectors. In *IMEC-IFIP Int'l Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111-128, November 1989.

[Gup92] A. Gupta. Formal Hardware Verification Methods: A Survey. In R.K. Brayton, E. M. Clarke and P.A. Subrahmanyam, editors, *Formal Methods in System Design*, pages 5-92. Kluwer Academic Publishers, October 1992.

[PH90] J. Hennessy, and D. Patterson, *Computer Architecture A Quantative Approach*, Morgan Kaufman, 1990.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576-580, 1969.

[Hoa72] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271-281, 1972.

[Hun85] W. Hunt, *FM8501: A Verified Microprocessor*. University of Texas, Austin, Tech. Report 47, 1985.

[Joy88] J. Joyce. Formal Verification and Implementation of a Microprocessor. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 129-157. Kluwer Academic Publishers, 1988.

[Koh78] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw Hill, 1978.

[Seg87] R. Segal. *BDSYN: Logic Description Translator*. UC Berkeley, UCB ERL Memo No. M87/33, May 1987.

[SSMS92] E.M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K.Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *Proceedings of the Int'l Conference on Computer Design: VLSI in Computers and Processors*. pages 328-333, October 1992.

[TSBS90] H.Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDDs. In *Proc. of Int'l Conference on Computer-Aided Design*, pages 130-133, November 1990.