Automatic Verification of Pipelined Microprocessors

Vishal Bhagwati Srinivas Devadas
Research Laboratory of Electronics
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139 USA

Abstract - We address the problem of automatically verifying large digital designs at the logic level, against high-level specifications. In this paper, we present a methodology which allows for the verification of a specific class of synchronous machines, namely pipelined microprocessors. 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.

We use symbolic simulation of the specification and implementation to verify their functional equivalence. We characterize the pipelined and unpipelined microprocessors 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. We show that 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.

1 Introduction

We address the problem of automatically verifying large digital designs at the logic level, against high-level specifications. The work described in this paper focuses on verification of a class of synchronous machines, namely *pipelined microprocessors*.

Procedures for verifying strict input/output equivalence between two Finite State Machines (FSMs) were introduced in [CBM89]. This involved exhaustively traversing the State Transition Graph of the product of the two machines, using implicit state enumeration techniques. Recently, these procedures were extended to allow for differences in the input/output behavior of the specification and implementation [AAD91]. A relation is established between the input/output behavior of the implementation and specification. 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.

Theorem-proving techniques (such as in [Hun85], [Coh88] and [Joy88]) are more powerful than the approach of [AAD91], however they typically require extensive user interaction. These techniques contain the best-known examples of formally verified processors, but are extremely simple designs, which are generally unpipelined. Human-guided theorem provers described in [Ros92] and [SB90] have been successful in verifying pipelined processors, but in these cases, either the processor was extremely simple or a large amount of labor was required. A much more automatic technique is described in [LC91], but is not applicable to pipelined microprocessors. [Cor93] describes a program for analyzing logical expressions which is used for verifying a non-pipelined processor, but the problems of specifying pipelined processors are not addressed in this work. A method for verification of pipelined hardware is described in [BK89], which uses an *abstraction function* to map the state of a pipelined machine to an abstract unpipelined state.

In our 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.

Our strategy of verifying the functionality of the implementation against that of the specification involves implicit state enumeration techniques as described in [CBM89]. We characterize the pipelined and unpipelined microprocessor 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. We show in Section 3 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. This makes our methodology viable for large digital systems with complex pipelines.

This paper is organized as follows. Section 2 contains preliminaries on string function relations, and symbolic simulation of synchronous machines. In Section 3, we characterize microprocessors as definite machines and state some of their properties that are essential for verification purposes. Section 4 contains the methodology used for verifying the pipelined processor implementation against the unpipelined specification. In Section 5, we present experimental results. In Section 6, we conclude with a summary of our work and give directions for future work in this area.

2 Preliminaries

2.1 String Function Relations

This section 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. Synchronous machines realize a certain string function, first introduced in [Bro89]. They map a sequence (string) of input values into a sequence of output values.

With Σ as alphabet, the set of strings, Σ^*, is defined as the set of all finite concatenations of elements of the alphabet. We use variables u, v, ... for characters, and x, y, ... for strings. The empty string is denoted by ε. String functions are functions from Σ^* to Σ^*. Useful operations on strings include:

- the concatenation operation, written as ., which concatenates two strings (or a string and a character),
- a length operation written as $| \, |$, where $|x|$ is the length of string x,
- an operation written as $\uparrow$, which takes a character and a number n, and returns n repetitions of the character,
- and an operation written as $\downarrow$, 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 $(x\downarrow i).....(x\downarrow j)$.

An implementation is considered correct with respect to a specification, if a certain relation holds between the two machines. The relation verified in our research is similar to the "don't care times" relation, β, which is defined in [AAD91]. To define the β-relation, a function *Relevant* has to be defined first.

Definition 2.1.1 (Relevant): $\Sigma^* \times \mathbf{B}^* \rightarrow \Sigma^*$:

$$\begin{aligned} &\text{Relevant}(\varepsilon,\varepsilon) = \varepsilon \\ &\text{Relevant}(x.u,y.v) = \text{Relevant}(x,y) \quad ,v = 0 \\ &\qquad\qquad = \text{Relevant}(x,y).u \quad ,v = 1 \end{aligned}$$

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 2.1.2 (β-relation):

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

$$F \, \beta_{H,n} \, G \Leftrightarrow \forall x \in \Sigma^* \ s.t. \ |x| \geq n,$$
$$Relevant\ (F(x),\ R_{0\uparrow n}\ o\ H(x)) =$$
$$G\ (Relevant\ (x\downarrow 1...(|x|-n),\ H(x\downarrow 1...(|x|-n)))).$$

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.

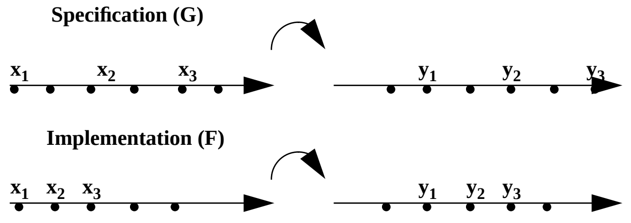

Fig 1. **The β-relation between *F* and *G***

2.2 Binary Decision Diagrams and Symbolic Simulation

We verify the β-relation between the implementation and the specification by traversing the state space and enumerating the states for each machine. We uses *symbolic simulation* of logic using Reduced Ordered Binary Decision Diagrams (ROBDD's) for this purpose. A more detailed description of ROBDD's and their application to combinational logic verification can be found in [Bry86].

For sequential machines, the state transition relation can be obtained from the state transition graph and can be expressed in a canonical form using ROBDD's. Given a current set of states, image computations are done using the transition relation for determining next set of states for the machine. This method traverses the state space of the machine in a breadth-first manner. We will not elaborate on sequential machine state traversal and enumeration here; a more detailed treatment of this topic is given in [CBM89].

3 Microprocessors as Definite Machines for Verification

3.1 Definite Machines

Definition 3.1.1 (Definite Machine):

A sequential machine *M* is called a *definite machine* of order μ if μ is the least integer, such 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. If a machine is μ-definite, it is also of finite memory of order equal to or smaller than μ. Figure 2 shows a canonical realization of a μ-definite machine. A more detailed treatment of definite machines is given in [Koh78].

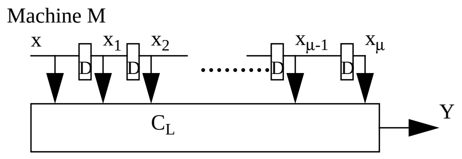

Fig 2. **Canonical Realization of a μ-definite machine**

3.2 Verification Properties of Definite Machines and Microprocessors

3.2.1 Verification of Definite Machines

Theorem 3.2.1.1 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.

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. ❑

3.2.2 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, it performs the operation and changes the state of the machine, which 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.

3.2.3 β-relation for Verification of Definite Machines

Theorem 3.2.3.1 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 3 shows the logic transformation $S_{G'}$ and the string function S_{H1} to filter the relevant outputs for the unpipelined machine S_G. Figure 4 shows the logic transformation $S_{F'}$ and the string function S_{H2} to filter the relevant outputs for the pipelined machine S_F. 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. 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. ❑

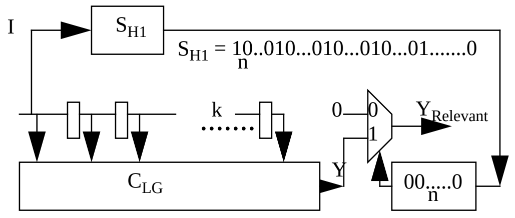

Fig 3. **Logic Transformation $S_{G'}$ to Unpipelined *k*-definite machine S_G**

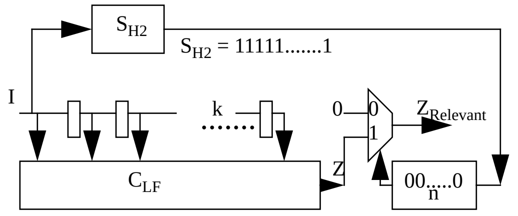

Fig 4. **Logic Transformation $S_{F'}$ to Pipelined *k*-definite machine S_F**

3.2.4 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*, $1 \le i \le k$, 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*, $1 \le q \le (i-1)$ modifies the state of the machine (i.e., writes registers, modifies program counter etc., in the case of a microprocessor) in stage *j*, $q < j \le k$, 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 3.2.4.1 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 3 and Figure 4 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 4)*

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 x^{th} instruction is a control transfer instruction with $i\text{-}1$ delay slots, then

1. InputUnpipelinedMachine$_{1....x}$ = InputPipelinedMachine$_{1....x}$,
2. InputUnpipelinedMachine$_{x+1......k}$ = InputPipelinedMachine$_{x+i,........max\ (k,\ j+i\text{-}q)}$, and
3. Outputs for InputPipelinedMachine$_{x+1.....x+i\text{-}1}$ 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 $i\text{-}1$ delay slots, there would be $p^{i\text{-}1}$ possible instruction sequences of length $i\text{-}1$. Therefore the number of possible instruction sequences becomes $x{*}p^k{*}p^{i\text{-}1} + p^k$, 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

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

❑

3.2.5 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. The possible data hazards are Read After Write (*RAW*), Write After Read (*WAR*), and Write After Write (*WAW*). 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 3.2.5.1 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 3.2.4 can be used to verify the two k-definite machines. ❑

4 Verification of Pipelined Microprocessors using Symbolic Simulation

Our methodology of verification of pipelined microprocessors is incorporated into *sis* [SSM92], 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.

4.1 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 5.

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 $2k\text{-}1$ 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 $k(i\text{-}1)+1$, and is an input in cycle $k(i\text{-}1)+2$. At the $(k(i\text{-}1)+2)^{th}$ 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, $1 \le i \le k$. 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.

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 $k\text{-}1$ cycles.

Section 4.3 contains a discussion on variables to be observed, and verification of the BDD formulae of these variables.

```
Verify(U,P,simulationInfo) {
/* U is the Unpipelined Network,
   P is the Pipelined Network */
 Compute input specification function for P;
 Compute output filtering function for P;
 Compute transition relation for P;
 Simulate P for 2k-1 cycles;
 If in a simulation cycle, the output filtering
function is 1, sample the specified variables and
add their formulae to the array varP;
 Compute input specification function for U;
 Compute output filtering function for U;
 Compute transition relation for U;
 Simulate U for k^2 cycles;
   If in a simulation cycle, the output filtering
function is 1, sample the specified variables and
add their formulae to the array varU;
 for (i=1, i ≤ K; i=i+1) {
   for (j=1, j ≤ NUM_VARS; j=j+1) {
   verifyBddFormulae(varU[i][j], varP[i][j]);
   if not equal then the two machines are not func-
tionally equivalent and exit; } } }
```

Fig 5. Algorithm for verifying the *functional equivalence* of the pipelined and unpipelined microprocessors

4.2 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 that 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.

4.3 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:

- General Purpose Registers,
- Instruction Address Register (the Program Counter PC),
- Memory Location Contents,
- Address to Register File and Memory for Read/Write,
- Instruction Register,
- 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.

4.4 Verification of Complex Pipelined Structures

The methodology for verification of simple pipelined microprocessors can be extended to verify the functionality of microprocessors with complex control to handle interrupts, traps, exceptions and also dynamically scheduled pipelines. This is described in detail in [Bhag93]. A method to verify superscalar pipelined processors is also described.

5 Experimental Results

To demonstrate the feasibility of our methodology, we performed experiments on two pipelined microprocessor designs. The first design, VSM, is a simple RISC pipelined microprocessor. The VSM is described in [Bhag93]. The salient features of VSM are as follows:

- Five 13-bit single format instructions.
- 4-stage static pipeline.
- Eight 3-bit general purpose registers.
- 3-bit ALU operations.
- One delay-slot after the branch instruction,
- 5-bit Instruction Address Register (PC).

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].

The second design, Alpha$_0$, is a subset of the DEC-Alpha™ microprocessor. The Alpha$_0$ is described in [Bhag93]. 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:

- Load/Store RISC architecture,
- Sixteen 32-bit fixed format instructions,
- 5-stage static pipeline,
- Thirty-two 4-bit registers,
- 4-bit ALU operations,
- One delay-slot after each control transfer instruction,
- 5-bit Instruction Address Register.

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.

6 Summary

The main contribution of this paper is that a sound methodology for verification of pipelined processors has been developed. The primary computation cost in this and other similar methods is BDD manipulation, and cannot be directly applied 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 Section 5), non-essential combinational logic that increases BDD size can be discarded for improved efficiency.

The strategy described in this paper 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.

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.

Acknowledgements

This research was supported in part by the Advanced Research Projects Agency under contract N00014-91-J-1698, and in part by a NSF Young Investigator Award with matching funds from Mitsubishi and IBM Corporation.

References

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

[Bhag93] V. Bhagwati. *Automatic Verification of Pipelined Microprocessors.* S.M. Thesis, MIT, 1993.

[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.

[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.

[Cor93] F. Corella. Automatic high-level verification against clocked algorithmic specifications. In *Proceedings of the IFIP WG10.2 Conference on Computer Hardware Description Languages and their Applications*, Ottawa, Canada. April 1993. Elsevier Science Publishers B.V.

[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.

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

[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.

[LC91] M. Langevin and E. Cerny. Verification of processor-like circuits. In P. Printetto and P. Camurati, editors, *Advanced Research Workshop on Correct Hardware Design Methodologies*, June 1991.

[Ros92] A. W. Roscoe. Occam in the specification and verification of microprocessors. *Philosophical Transactions of the Royal Society of London, Series A: Physical Sciences and Engineering*, 339(1652):137-151, Apr. 15, 1992.

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

[SSM92] 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.

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