

# Efficient Microprocessor Verification using Antecedent Conditioned Slicing

Shobha Vasudevan  
Computer Engg Research Center  
University of Texas at Austin  
shobha@cerc.utexas.edu

Vinod Viswanath  
Intel Corporation  
Austin, TX  
vinod.viswanath@intel.com

Jacob A. Abraham  
Computer Engg Research Center  
University of Texas at Austin  
jaa@cerc.utexas.edu

## Abstract

We present a technique for automatic verification of pipelined microprocessors using model checking. Antecedent conditioned slicing is an efficient abstraction technique for hardware designs at the Register Transfer Level (RTL). Antecedent conditioned slicing prunes the verification state space, using information from the antecedent of a given LTL property. In this work, we model instructions of a pipelined processor as LTL properties, such that the instruction opcode forms the antecedent. We use antecedent conditioned slicing to decompose the problem space of pipelined processor verification on an instruction-wise basis. We pass the resulting smaller, tractable problems through a lower level verification engine.

We thereby verify that every instruction behaves according to the specification and ensure that non-target registers are not modified by the instruction. We use the SMV model checker to verify all the instruction classes of a Verilog RTL implementation of the OR1200, an off-the-shelf pipelined processor.

## 1. Introduction

Formal verification of a pipelined microprocessor is a challenging and hitherto unsolved problem. Theorem proving approaches to processor verification [15],[20] have produced an outstanding body of research related to verification of complex “modern” processors. In an industrial setting, however, due to the preference for automated techniques, formal verification is limited to isolated functional units. Simulation based verification (testing) techniques are used to validate large parts of a modern industrial processor design.

Model checking [8] and other Boolean level techniques (like SAT, BDD or STE based techniques) although automatic, suffer from inherent limitations like high memory requirements and restricted applicability. As such, these techniques have not been applied to processor verification as much as deductive verification techniques. It is desirable to use abstraction techniques to reduce the complexity of the verification problem, to leverage the automatic power of these Boolean level verification techniques.

In [22],[23], an abstraction technique called *antecedent*

*conditioned slicing* was introduced. Antecedent conditioned slicing is a static analysis technique that uses information from the antecedent of an LTL property [14] to prune the model checking state space. When applied to Register Transfer Level (RTL) designs described in Hardware Description Languages (HDLs), antecedent conditioned slicing shows significant performance gains.

In this paper, we apply antecedent conditioned slicing to automatically decompose the pipelined microprocessor verification problem into simpler sub-problems that can be handled by Boolean level engines. In the context of processor verification, this automatic abstraction technique provides a conduit for tapping into the efficiency of these state-of-the-art lower level engines. Our technique involves an instruction-wise decomposition. An instruction’s execution is represented by an LTL property, with the instruction opcode as the antecedent. We prune the HDL description of the processor using antecedent conditioned slicing. We pass the resulting antecedent conditioned slice through a model checker. We thus check if every instruction’s operation is as specified. We also prove certain other lemmas to ensure that there is no interference from incorrect results of other instructions. We thereby verify the correctness of the interactions among the instructions of a pipelined processor. We present our experiments on an RTL implementation of the OR1200, an off-the-shelf pipelined microprocessor. We show verification results of all instruction classes of this processor using our technique. We use the SMV [16] model checker as the lower level engine to show our results.

The principal contributions of this paper are:

- We introduce an automatic decomposition technique for microprocessor verification using antecedent conditioned slicing. The decomposition can be used for splitting the processor verification problem space into more tractable problems.
- Our algorithm provides a channel to leverage the power of automatic Boolean level engines for processor verification. Although we have used model checking as the lower level engine, our RT-level static analysis technique is intended for synergistic use with SAT, BDD and/or STE based engines.

- We describe a verification technique that works with any generic processor design described in RTL, as opposed to a high-level model of the processor. This, as stated in [2] is a very relevant issue in contemporary hardware verification environments.
- We demonstrate an alternative notion of verification coverage, by verifying all the instructions in a processor, as opposed to traditional notions of checking a pipeline. In doing so, we provide a generic framework for verification of single instruction issue, multi-stage pipelined processors in RTL.

The organization of the paper is as follows. Section 2 gives an overview of slicing techniques and explains antecedent conditioned slicing. Section 3 describes the algorithm for instruction verification using antecedent conditioned slicing and gives a sketch of the proof methodology. In Section 4, we discuss the OR1200 processor model. In the rest of Section 4 we detail the proof methodology for that processor and present the experimental results of model checking the OR1200 instruction classes. Section 5 describes related work in this field and provides a qualitative comparison to relevant work. We discuss the limitations and applications of our technique and conclude with Section 6.

## 2. Background

We explain some background concepts necessary for understanding our technique.

### 2.1. Slicing Techniques

Slicing [27] is a program transformation involving statement deletion that preserves some projection of the semantics of the original program. The aspect of the program that must be preserved is application specific, and is captured by the slicing criterion.

A *slicing criterion* of a program  $P$  with an input alphabet  $\Sigma$ , is a pair  $\langle i, V \rangle$  such that  $i$  is a point in  $P$  and  $V \subseteq \Sigma$ . A set of statements  $I_s$  is said to *affect* the values of  $V$  at  $i$  in a given slicing criterion  $\langle i, V \rangle$ , if  $I_s$  defines a subset of  $V$  that is used in  $i$ . A *static slice*  $S$  of a program  $P$  on a slicing criterion  $\langle i, V \rangle$  is a subset of the statements of  $P$  that might affect the values of  $V$  at  $i$ .

Static slicing can result in very large slices. Conditioned slicing [7] augments static program slicing by introducing a condition that specifies the initial set of states in the slicing criterion.

Let  $\Sigma$  be the input alphabet to a program  $P$ . Let  $C$  be a first order predicate logical formula on the variables in  $\Sigma$ . A *conditioned slicing criterion* is a triple  $\langle C, i, V \rangle$ , where  $i$  is a program point, and  $V \subseteq \Sigma$ . A *conditioned slice* of a program  $P$  on a conditioned slicing criterion  $\langle C, i, V \rangle$  consists of all the statements and predicates of  $P$  that might affect the values of the variables in  $V$  at  $i$ , when the condition  $C$  holds true.

### 2.2. Antecedent Conditioned Slicing

In [22], conditioned slicing was extended to HDLs and introduced as an abstraction for verification. The details of this process as well as the correctness proofs for these abstractions can be found at [23].

Conditioned slicing is used in the context of verification of hardware designs described in Verilog HDL. The aim is to reduce the design's state space, by slicing away parts of the design (HDL statements) that are irrelevant to the property being verified.

For properties expressed as LTL formulas, the antecedent essentially specifies the set of initial states of interest. *The antecedent, therefore, forms the condition in the slicing criterion.* All the statements that would get executed when the antecedent is true (or the condition is satisfied) are included in the slice. The statements on the paths that cannot get executed when the antecedent is true, are removed. The reduced program still preserves its behavior *with respect to the property being checked.* The property preserving abstraction thus created, is called an *antecedent conditioned slice*.

### 2.3. Computing antecedent conditioned slices

We illustrate antecedent conditioned slicing with an example. The reader can skip this example without any loss in continuity. Figure 1(a), shows a typical Verilog state machine.

Let  $h1 = [G((insn == add) \Rightarrow XX(res == a + b))]$ . Now, the static slice will retain all the statements that define the variable  $insn$  along with  $add$ ,  $res$ ,  $a$  and  $b$ . However, in the antecedent conditioned slice, we prune the statements that define  $insn$ , but do not appear in the antecedent, as explained in Subsection 2.2. In the antecedent conditioned slice, we include all the statements that correspond to the conditioned slice with respect to the criterion  $\langle (insn = add), end, \{insn, res, a, b\} \rangle$ . Since the property extends across three cycles, we include all the statements that will be executed in the next two clock cycles when the antecedent is true. The resulting antecedent conditioned slice is shown in Figure 1(b).

## 3. Processor Verification using Antecedent Conditioned Slicing

In pipelined processor verification, Burch and Dill's [6] pioneering technique reasons about the entire state of the pipeline using an abstraction function. Since the abstraction (flushing) function is very complex for any reasonably sized processor, there has been subsequent work in refinement of the abstraction function, in order to create more tractable parts of it.

A processor's microarchitecture is described in terms of its instruction set. It is therefore intuitive to reason about the

<pre> always @ (clk) begin   case (insn)     f_add: dec = d_add;     f_sub: dec = d_sub;     f_and: dec = d_and;     f_or:  dec = d_or;   endcase end </pre>	<pre> always @ (clk) begin   case (dec)     d_add: ex = e_add;     d_sub: ex = e_sub;     d_and: ex = e_and;     d_or:  ex = e_or;   endcase end </pre>	<pre> always @ (clk) begin   case (ex)     e_add: res = a+b;     e_sub: res = a-b;     e_and: res = a&amp;b;     e_or:  res = a b;   endcase end </pre>
(a) Verilog code with state machine		
<pre> always @ (clk) begin   case (insn)     f_add: dec = d_add;   endcase end </pre>	<pre> always @ (clk) begin   case (dec)     d_add: ex = e_add;   endcase end </pre>	<pre> always @ (clk) begin   case (ex)     e_add: res = a+b;   endcase end </pre>
(b) Antecedent conditioned slice for $h1$ .		

**Figure 1. Example Verilog code and antecedent conditioned slice for the LTL property  $h1$ .**

behavior of individual instructions while verifying the processor. In contrast to Burch and Dill, we do not reason about the entire state of the pipeline, but about individual instructions. This approach was also used by Jhala and McMillan in [13]. Since every instruction is implemented deterministically, with finite resources, this is tantamount to verifying many small finite state machines.

We achieve this instruction-wise decomposition using antecedent conditioned slicing. In order to apply antecedent conditioned slicing to processor verification, we model an individual instruction’s behavior as an LTL property whose antecedent corresponds to the opcode of instruction being verified<sup>1</sup>. An instruction’s behavior is expressed as a property of the form  $[G(a \Rightarrow X^n c)]^2$ . For a pipelined processor, the antecedent at every time step involves the stages of the pipeline that an instruction needs to pass through to get the required output. Antecedent conditioned slicing, therefore eliminates the portion of the system behavior that is irrelevant to the instruction. Only those portions of the pipeline that are affected during the lifetime of an instruction, are retained. The resulting single instruction machine abstraction is used to verify the LTL property. Evidently, this abstraction provides a much smaller and simpler system to the model checker. We check every instruction’s behavior using the above approach. The property checks if the instruction being verified performs the specified operations with respect to its intended destination (target) registers.

However, in order to guarantee correctness of the pipeline processor, we also need to consider interaction between instructions. This is done by proving ancillary lemmas (expressed as temporal properties) that check (a) if the pipeline control signals like stalls, freeze, data forwarding, program counter, *etc.*, function according to specification; (b) if the instruction being verified does not modify non-target registers. This is to ensure the correctness of subse-

quent instructions in the pipe. We prove these lemmas once for the entire processor design, using a model checker. We discuss these lemmas in detail in Subsection 4.1.

### 3.1. Algorithm for instruction verification

Let  $P$  be a Verilog program describing a pipelined processor microarchitecture, with pipeline depth  $n$ .

We outline the algorithm for verifying an instruction  $I$  of processor  $P$  using our technique in Figure 2.

Let  $h = [G(I \Rightarrow X^k R)]$  be an LTL formula. Let  $I$ , the instruction word, be the antecedent of  $h$ . Let  $k \leq n$  be the number of cycles<sup>3</sup> taken to execute  $I$ .  $R$  represents the expected result of executing instruction  $I$ , in terms of target register values.

The procedure *insn\_check()* computes the antecedent conditioned slice over the specified  $k$  time steps. For the first time step  $t = 0$ , the antecedent conditioned slice consists of the set of statements that would be executed when  $I$  holds. For any future time step  $t$ , the antecedent conditioned slice contains the set of statements that would be executed  $t$  time steps later, when  $I$  holds in  $t = 0$ . The final antecedent conditioned slice is recursively defined as the entity obtained by the union of the antecedent conditioned slices obtained over successive time steps, denoted by the symbol  $\bigsqcup$ .

This simply means that all the statements that would get executed when the antecedent is true in the current time step, are retained, along with all the statements in future time steps that would be executed when the antecedent is true. The resulting antecedent conditioned slice is then statically sliced with respect to the slicing criterion  $\langle e, V \rangle$ . This ensures that all the statements that affect the variables  $V$  in the conditioned program are retained and the others are deleted. The procedure *get\_static\_slice()* obtains a static program slice. Details of the static slicing algorithm for HDLs can be found in [9], [24].

<sup>1</sup>We capture the desired behavior of the antecedent by iteratively refining it using the counterexample generated by a model checker.

<sup>2</sup>We also allow liveness properties in our property specification language. However, verification related properties are typically safety properties.

<sup>3</sup>In Verilog programs describing sequential hardware circuits, a clock is explicitly modeled in the design. Successive time steps are, therefore, according to the progression of this clock (cycles). We, therefore, use “time step” interchangeably with “cycle”.

```

single.insn.check ( $P$  : Verilog program,  $I$  : instruction,  $h$  : LTL
property)
begin
   $h = [G(I \Rightarrow X^k R)]$ 
   $insn\_check(P, h)$ 
end

insn.check ( $P, h$ )
begin
   $P_{-1} = \phi$ ;
  for every time step  $t \geq 0$ , while  $t \leq k$ 
  begin
     $P_t = P_{t-1} \parallel get\_conditioned\_slice(P, t, \langle I, end, V_h \rangle)$ ;
  end
   $P_s = get\_static\_slice(P_k, \langle end, V \rangle)$ 
  return  $model\_check\_result(P_s, h)$ ;
end

get.conditioned.slice ( $P, t, \langle C : condition, e : program\ point, V : set$ 
of variables  $\rangle$ )
begin
  for every statement  $x$  in  $P$  until  $end$ 
  begin
    if ( $C$  is true implies  $x$  is executed  $t$  cycles later)  $mark_x = T$ 
    else if ( $C$  is false implies  $x$  is executed  $t$  cycles later)
       $mark_x = F$ 
    else if ( $x$  does not ever depend on a truth value of  $C$ )
       $mark_x = X$ 
  end
  for every path  $\rho$  in  $P$ 
  begin
    for every statement  $x$  along  $\rho$ 
    begin
      if ( $mark_x = F$ )  $slice(\rho)$ 
      else  $retain(x)$ 
    end
  end
  return  $P$ 
end

```

**Figure 2. Algorithm for verifying an instruction  $I$  using antecedent conditioned slicing.**

The procedure  $model\_check\_result()$  shows the step where the antecedent conditioned slice is then passed through a model checker along with the property  $h$ . In case the property is not verified, a counterexample is returned.

The procedure  $get\_conditioned\_slice()$  computes the conditioned slice for a given time step  $t$  and a conditioned slicing criterion  $\langle C, e, V \rangle$ . A statement is marked  $T$  in the slice if it is executed  $t$  time steps later from the time step when the condition  $C$  holds true. A statement is marked  $F$  if it is executed  $t$  time steps later from the time step when  $C$  is false. If the statement does not, in any future, depend on the truth value of  $C$ , it is marked  $X$  in the slice. In order to determine the statements that are executed in future time steps, an analysis of the guards (control paths) in the program control flow graph is done. In order to determine the truth value of the antecedent at a given statement, a simple structural check is done. Since the antecedent in this case is

the instruction, which is an input, this check is sufficient. In the case where the antecedent involves signals that are assigned within the program, this check would require a more complex evaluation procedure.

After all statements have been marked with the truth value of the antecedent, all the paths in the program are traversed. A path where all the statements are marked  $F$  is sliced, while a path with any of the statements marked  $T$  or  $X$  is retained. In the case of processor designs, the truth value of the antecedent does not change along a path. This is because the antecedent is the instruction word itself. Since the antecedent (instruction) does not change for any given path, the path traversal step can be optimized to be more efficient. We have given a very high level overview of this algorithm. The formal definitions and details of applying conditioned slicing to HDLs are available in [22].

#### 4. Verification of the OR1200 processor

In order to prove the correctness of the pipeline, we need to prove that each instruction is correct in the presence of other instructions, both ahead of (older) and behind (younger) in the pipe. We prove this result inductively. We assume all instructions older than the current one are correct in the presence of other instructions. The base case is trivially true in the reset state when there are no older instructions. We need to prove that the current instruction is correct in the presence of instructions younger than itself in the pipeline. This proof is broken down into three parts. In the first part, we show that the current instruction is correct, *i.e.* it updates the correct target register(s) with the functionally correct computation. In the second part, we prove that the control logic of the pipe *i.e.* the flush, freeze, stall, program counter and data forwarding (bypass) logic function according to specification. In the third part, we prove that the current instruction does not modify non-target registers in the register file (or non-target memory locations in the case of STORE instructions). These three lemmas together prove the inductive step. We now detail our proof methodology, when applied to a pipelined processor microarchitecture.

We use the OpenRisc 1200, a publicly available processor for our experiments. The specification manual of the OR1200 is at [10] and the source code of its implementation in Verilog RTL can be obtained from [17]. The OR1200 is a 32-bit scalar RISC with Harvard microarchitecture, 5 stage integer pipeline, virtual memory support (MMU) and basic DSP capabilities. OR1200 is intended for embedded, portable and networking applications.

OR1200 implements 32 general-purpose 32-bit registers. Special purpose registers (SPRs) of all units are grouped under 32 groups. The load/store unit (LSU) transfers all data between the general purpose registers and the CPU's internal bus.

## 4.1. Proof methodology

We outline some details of the proof for an example instruction of the OR1200 processor. The `l.addc` (add with carry) instruction is specified in the instruction manual as follows.

```
rD[31:0] = rA[31:0] + rB[31:0] + SR[CY]
SR[CY] = carry
```

The instruction word is specified as given in [10].

**Correctness Criterion for a Single Instruction** Our correctness criterion is that given an LTL property whose antecedent corresponds to an instruction issue, and the consequent corresponds to the specified<sup>4</sup> output values, the LTL property should hold true on the processor’s microarchitecture model.

**Assumptions** We disable the reads and writes from the debug unit to the special purpose registers (SPRS) unit. We also disable pending interrupts. We do this by ensuring that these values do not get asserted in every cycle.

**Instruction Behavior** We describe an LTL property for the `l.addc` instruction. We obtain the desired antecedent by counterexample guided refinement. The constraints required to write an LTL property acceptable to the model checker are written as a conjunction with the instruction word. The instruction word is first given as an antecedent.

```
if ((icpu_dat_i[31:26]== 6'b 111000) &&
    (icpu_dat_i[9:8] == 2'b00) &&
    (icpu_dat_i[3:0] == 4'b0001))
```

The add opcode is specified in the higher order bits of the instruction word, and the lower bits correspond to the opcode for the ALU. When passed to the model checker, this property generates a counterexample that shows a necessity to specify that a pipeline freeze or a pipeline flush has not been issued. Similarly, we also specify that a restore from exceptions is not necessary or there is no need to force fetch a delay slot. The antecedent is refined using the following constraints:

```
if (!rst && !flushpipe && !no_more_dslot &&
    !rfe && if_freeze)
```

These two constraints capture the antecedent for the instruction fetch phase. We specify the antecedents for the next (decode) cycle, by stating the following:

```
wait(1);
if (!rst && !flushpipe && !id_freeze &&
    !ex_freeze)
    wait(1);
```

The constraints, as indicated by the counterexample, need to be specified for every cycle.<sup>5</sup> For instance, negating the signal `id_freeze` disables the freezing of the decode

<sup>4</sup>This refers to the specification manual of the processor, where each instruction’s output values in target registers is given.

<sup>5</sup>The constraints required for refining the antecedent are external to our technique. Although this is a manual step, the resulting overhead is part of writing an acceptable property for the model checker, irrespective of our preprocessing technique.

unit in the decode cycle. When the decoded address reaches the register file, we latch the resulting operand and carry values in `a`, `b`, `c`. At the end of the writeback stage (5 cycles), we capture the result from the register file in `wbres` and compare it with the expected result. The consequent is written as follows:

```
assert addc: ((wbres == (a + b + c)));
```

**Other Instruction Classes** We verify all classes of instructions like Arithmetic Logic Unit (ALU) instructions, shift rotate (SHF/ROT) instructions, load store unit (LSU) instructions, branch and jump instructions, multiply and accumulate unit (MAC) instructions. The corresponding LTL properties can be found at [1]. For loads, we verify that the data is read from the correct address and loaded into the specified registers. For stores, we verify that the data specified in the source register is written to the correct target address. We do not explicitly verify the memory array itself.

**Lemmas for Pipeline Correctness** We now need to prove the lemmas that ensure pipeline correctness, according to the second part of our proof. We check that the program counter is incremented correctly for all instructions. We check the branch and jump instructions separately. Proving this result for the other instructions is trivial. We also prove lemmas to ensure that the pipeline flush, freeze, the LSU, instruction fetch unit, debug unit and other stalls, and the data forwarding (bypass) logic perform correctly.

The third part of the proof is given by an additional lemma over the entire processor design that ensures that no incorrect values are written to other (programmer visible) registers in the register file. We disable some exceptions by not allowing illegal instructions. We also do not allow any stalls, freezes or flushing in the pipeline while proving this lemma.

There is only one write enable signal on the entire register file. There are two 32X32 dual-ported RAMs constituting the register file, one port is a read port and the other one is a write port. The `rf_we` signal decides when the register file is to be written. `rf_we` is asserted when an instruction writes back to the register file.

In order to prove that an instruction does not write incorrectly to any non-target register, we need to show that an instruction writes to the register file only during its write back stage, when it writes to its target registers.

```
assert lemmal:
((wbr == tr) &&
 ((reg_writeback_valid & rf_we) |
 (~reg_writeback_valid & ~rf_we)))
```

`reg_writeback_valid` represents the instructions that write back to the register file. For the instructions that write back to the register file, the write enable signal gets asserted in the cycle when the register being written (`wbr`) is the same as the target register (`tr`) in the instruction word We

also prove lemmas to ensure that the pipeline flush, freeze and stalls are performed correctly.

We have implemented our antecedent conditioned slicing algorithm in C++. We used our slicer to generate antecedent conditioned slices for each instruction. The slicing times taken by the slicer are in the order of 300 seconds.

The antecedent conditioned slices for every instruction were then given to the model checker. We have used our methodology to verify all the instruction classes of the OR1200. All experiments were run on a 3 GHz Intel Pentium 4 processor with 1GB RAM. We used the SMV model checker [16] with the abstraction refinement (absref3) option. This option utilizes SAT solvers and BDDs in combination to do the verification. The results of our experiments are shown in Table 1. On the original processor design, SMV, with the same options ran out of memory and did not finish the verification of the properties. We do not present these results, since all properties uniformly did not complete the verification process, even beyond 60 hours.

## 5. Related Work

There has been a significant amount of research in the field of pipeline processor verification as described in [3]. Most of these are variations of the Burch and Dill method [6].

Theorem proving techniques for processor verification have been widely researched. Sawada and Hunt [20] refined the Burch and Dill correctness criterion to verify a complex microprocessor with exceptions, stalls, interrupts etc using the ACL2 theorem prover. Velev and Bryant [25], [26] enhance the Burch and Dill method by efficiently using uninterpreted functions, equality etc. Hosabettu et al [11] use *completion functions* to represent the status of an instruction.

All these techniques try to verify complex processors with superscalar attributes. In order to demonstrate a correspondence between the implementation and the reference model, these techniques require complicated invariants whose construction requires expertise. In this work, we focus on completely automating the pipelined processor verification problem. We therefore show our results on a simple single instruction issue, multi-stage pipelined processor, that does not have interrupts, exceptions or other sophisticated features of a superscalar processor. In a sense, our work is orthogonal to the theorem proving efforts, as our drift is more toward automation, and less toward complex processor designs.

In recent times, Symbolic Trajectory Evaluation (STE) [5] techniques have been applied with considerable success to verify components of the Pentium 4 processor [4], [19], [21]. The STE and GSTE [12] techniques have been used to verify the floating point execution units in Intel's Pentium 4 processors. Although these techniques can also handle temporal properties of the forms we use in our technique,

they are limited by the inherent pitfalls of their scalability. Our decomposition strategy may enhance the applicability and scope of STE by providing it more feasible problems.

Patankar et al [18] use an instruction based decomposition technique similar to ours in combination with STE engines. However, this approach might not scale, since it involves creating detailed trajectory maps for all possible sequences of instruction flow in the Boolean domain.

Jhala and McMillan [13] used a compositional model checking approach for verifying pipelined processors. Their work, like ours, is oriented toward processor verification on an instruction by instruction basis. However, their technique requires significant manual intervention to decompose the proof using symmetry, temporal case splitting and abstract interpretation. Our antecedent conditioned slices are computed using a generic algorithm, making the decomposition process automatic.

An important difference between previous processor verification techniques and ours, is that we do not build our own processor models for the purposes of verification, but use a publicly available processor implemented in Verilog RTL.

## 6. Discussion and Conclusions

Processor verification is an excellent application for our abstraction technique. The antecedent, which is the instruction word in a single instruction machine, does not change through the duration of the property. Also, since the antecedent is an input, the computation required to determine the truth of the antecedent at every step is relatively less complex. This would explain the major performance benefits of using the technique for this application.

In its current form, our technique is most effective for single instruction issue, multi-stage pipeline processors, such as graphics and embedded processors. Our future work involves applying the technique to processors with more complex features, like out-of-order execution, register renaming, interrupts etc.<sup>6</sup>

## References

- [1] OpenRISC 1200 Properties. [http://www.cerc.utexas.edu/~shobha/OR1200\\_properties/](http://www.cerc.utexas.edu/~shobha/OR1200_properties/).
- [2] M. Aagaard, V. Ciobotariu, J. Higgins, and F. Khalvati. Combining equivalence verification and completion functions. In *Formal Methods in Computer-Aided Design (FMCAD 2004)*, 2004.
- [3] M. D. Aagaard, B. Cook, N. A. Day, and R. B. Jones. A framework for microprocessor correctness statements. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144, pages 433–448, 2001.
- [4] B. Bentley. Validating the intel pentium 4 microprocessor. In *Proceedings of the 38th conference on Design automation*, pages 244–248, 2001.

---

<sup>6</sup>Acknowledgement: We would like to thank Dr. Jason Baumgartner from IBM, Austin, for his insightful comments on earlier versions of this paper.

Instruction Class	Instructions	SMV Time	Memory Usage
ALU	l.add	25.65s	23796KB
ALU	l.sub	24.70s	24018KB
ALU	l.addc	25.14s	25865KB
ALU	l.addi	21.60s	19658KB
ALU	l.addic	17.84s	16554KB
ALU	l.xor	24.84s	24831KB
ALU	l.and	23.28s	21727KB
ALU	l.or	24.01s	22761KB
BRANCH	l.bf	132.63s	44281KB
BRANCH	l.bnf	139.47s	46350KB
BRANCH	l.j	57.36s	31969KB
BRANCH	l.jr	59.64s	35177KB
BRANCH	l.jal	54.98s	31073KB
BRANCH	l.jalr	54.09s	30094KB
BRANCH	l.cmov	159.64s	49831KB
MAC	l.mul	25.28s	22801KB
MAC	l.mulu	26.63s	30004KB

Instruction Class	Instructions	SMV Time	Memory Usage
COMPARE	l.sfeq	157.29s	51731KB
COMPARE	l.sfne	183.01s	53801KB
COMPARE	l.sfgt	194.43s	55352KB
COMPARE	l.sfge	206.39s	56904KB
COMPARE	l.sflt	201.10s	58146KB
COMPARE	l.sfle	275.97s	63112KB
LSU	l.ld	35.85s	29104KB
LSU	l.lws	33.91s	28873KB
LSU	l.lwz	35.27s	29567KB
LSU	l.sd	38.32s	30941KB
LSU	l.sw	39.30s	31365KB
SHF/ROT	l.sll	26.81s	23771KB
SHF/ROT	l.srl	27.83s	24865KB
SHF/ROT	l.sra	28.42s	30847KB
SHF/ROT	l.ror	27.93s	26919KB
SPRS	l.mfspr	226.97s	50696KB
SPRS	l.mtspr	212.27s	48627KB

**Table 1. Times taken in seconds by SMV for verifying antecedent conditioned slices for all classes of instructions of OR1200 (all instructions not shown). Memory usage shown is total memory used for the entire verification operation (including both SAT and BDD phases).**

- [5] R. E. Bryant, D. L. Beatty, and C. H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *Proceedings of the 28th conference on ACM/IEEE design automation*, pages 397–402, 1991.
- [6] J. R. Burch. Techniques for verifying superscalar microprocessors. In *DAC '96: Proceedings of the 33rd Annual Conference on Design automation*, pages 552–557, 1996.
- [7] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology Special Issue on Program Slicing*, 40:595–607, 1998.
- [8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.
- [9] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. In *Correct Hardware Design and Verification Methods*, pages 298–312, 1999.
- [10] D. Lampret *et al.* Openrisc 1000 architecture manual. [http://www.cerc.utexas.edu/~shobha/openrisc\\_arch3.pdf](http://www.cerc.utexas.edu/~shobha/openrisc_arch3.pdf), 2003.
- [11] R. Hosabettu, G. Gopalakrishnan, and M. K. Srivas. Verifying advanced microarchitectures that support speculation and exceptions. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 521–537, 2000.
- [12] C.H. Seger J. Yang. Generalized symbolic trajectory evaluation - abstraction in action. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, pages 70–87, 2002.
- [13] R. Jhala and K. L. McMillan. Microarchitecture verification by compositional model checking. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 396–410, 2001.
- [14] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107, 1985.
- [15] Panagiotis Manolios and Sudarshan K. Srinivasan. Refinement maps for efficient verification of processor models. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1304–1309, Washington, DC, USA, 2005. IEEE Computer Society.
- [16] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37(1-3):279–309, 2000.
- [17] OPENCORES. <http://www.opencores.org>.
- [18] V.A. Patankar, A. Jain, and R.E. Bryant. Formal verification of an arm processor. In *Twelfth International Conference On VLSI Design*, pages 282–287, 1999.
- [19] K. R. Kohatsu R. Kaivola. Proof engineering in the large: formal verification of pentium 4 floating-point divider. In *International Journal of Software Tools and Technology Transfer*, pages 323–334, 2003.
- [20] J. Sawada and W. A. Hunt Jr. Results of the verification of a complex pipelined machine model. In *Conference on Correct Hardware Design and Verification Methods*, pages 313–316, 1999.
- [21] T. Schubert. High level formal verification of next-generation microprocessors. In *Proceedings of the 40th conference on Design automation*, pages 1–6, 2003.
- [22] S.Vasudevan, E.A.Emerson, and J.A.Abraham. Efficient model checking of hardware using conditioned slicing. In *4th Int. Workshop on Automated Verification of Critical Systems*, 2004.
- [23] S.Vasudevan, E.A.Emerson, and J.A.Abraham. Improved verification of hardware designs through antecedent conditioned slicing. In *International Journal of Software Tools and Technology Transfer*, July 2006.
- [24] V. M. Vedula, J. A. Abraham, J. Bhadra, and R. Tupuri. A hierarchical test generation approach using program slicing techniques on hardware description languages. *Journal of Electronic Testing: Theory and Applications*, 19(2):149–160, 2003.
- [25] M. Velev. Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 28, 2002.
- [26] M. N. Velev and R. E. Bryant. Formal verification of superscale microprocessors with multicyle functional units, exception, and branch prediction. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 112–117, 2000.
- [27] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, 10(4):352–357, 1984.