Microprocessor Verification using RT-Level Static Analysis Techniques

Shobha Vasudevan
Computer Engineering 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 Engineering Research Center
University of Texas at Austin
jaa@cerc.utexas.edu

Jason Baumgartner
IBM Enterprise Systems Group
Austin, TX
jasonb@austin.ibm.com

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 into 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 by instruction basis. We pass the resulting smaller, tractable problems through several lower level verification engines.

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 several Boolean level verification engines that use different algorithms 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 [16],[21] 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 [24],[25], 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 [15] 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. We had provided a high-level, conceptual algorithm in [24] for antecedent conditioned slicing. In this paper, we provide a detailed version of the algorithm that is closer to its implementation. The algorithm, in this form, has not been presented before. The antecedent conditioned slicing algorithm in its current form, provides a conduit to its application to processor verification.

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 channel for tapping into the efficiency of these state-of-the-art lower level engines. Our technique involves an instruction-wise decomposition (instruction slicing). In this paper, we present a revised algorithm for instruction slicing. We also discuss in detail, the efficacy and expediency of applying antecedent slicing to instruction slicing for processor verification.

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 provide an argument for establishing the correctness of pipelined processor verification using our approach.

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. In a previous version of this work, we had used the SMV [17] model checker as the lower level engine to show the results of our technique. In this work, we demonstrate the efficiency of our RT-level static analysis technique using several other Boolean level verification engines. The verification algorithms applied in the lower level engines include counterexample guided localization refinement [14], k-induction based SAT solvers [23], and STE [5]. The tremendous performance gains of using our static analysis technique, as compared to the inability of the lower level engines to verify the original processor model in reasonable time, stands testamentthe efficacy of

our technique.

The principal contributions of this paper are:

- We introduce an automatic decomposition technique for microprocessor verification using antecedent conditioned slicing. We present a detailed version of the antecedent slicing algorithm, as well as its extension to instruction slicing for processors. 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. We have shown the benefits of using our RT-level static analysis technique synergistically with SAT, BDD and STE based engines, using different verification algorithms.

- 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. We provide an argument for pipeline correctness using this approach. 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 novel look at the antecedent conditioned slicing algorithm, and provides a detailed explanation with an example. Section 3 describes the the algorithm for instruction verification using antecedent conditioned slicing and gives

a sketch of the proof methodology. Subsection 3.2 discusses how processor verification is an excellent application for the antecedent conditioned slicing technique. In Subsection 3.3, we provide an intuition for the correctness of our notion of pipelined processor verification. 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 verifying the OR1200 instruction classes using different Boolean level verification algorithms. 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. Revisiting Antecedent Conditioned Slicing

We explain some background concepts necessary for understanding our technique.

2.1. Background

Slicing [29] 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 Σ, 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 Σ be the input alphabet to a program P. Let C be a first order predicate logical formula

on the variables in Σ. 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 [24], 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 [25].

We give a brief intuition of the abstraction technique here. 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*. A property preserving, precise abstraction is thus created on the basis of conditioned slicing, called an *antecedent conditioned slice*.

Antecedent conditioned slices are powerful abstractions that give significant performance benefits over static program slicing [9] when applied to HDL verification. We describe them in detail in the next section.

2.3. Computing antecedent conditioned slices

The property language permits LTL properties h of the form

$$G(a \Rightarrow c), G(a \Rightarrow X^{=n}c), G(a \Rightarrow Fc)^k$$

where a and c are propositional formulas. $X^{=n}q$ means at distance $n \geq 0$, q holds, *i.e* $XXX \ldots n$ times. k represents the bound for bounded liveness properties.

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 2.4. 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).

2.4. Algorithm for antecedent conditioned slicing

In [24], we presented the algorithm for antecedent conditioned slicing at a conceptual level. We present a more detailed version of the algorithm that provides a platform for the instruction slicing algorithm for processor verification. We show the generic version of this algorithm, and discuss why processor verification is an excellent application for this technique.

```
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
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
always @ (clk) begin
 case(ex)
   e_add: res = a+b;
   e_sub: res = a-b;
   e_and: res = a&b;
   e_or:  res = a|b;
 endcase
              end
end
```

(a) Verilog code with state machine

```
always @ (clk) begin
case(insn)
  f_add: dec = d_add;
endcase
              end
always @ (clk) begin
 case(dec)
   d_add: ex = e_add;
 endcase
              end
always @ (clk) begin
 case(ex)
   e_add: res = a+b;
 endcase
              end
end
```

(b) Antecedent conditioned slice for $h1$

Figure 1. Example Verilog code with a state machine, and antecedent conditioned slice for the LTL property $h1$.

```
antecedent_conditioned_slice (P, G(a => X^{=n} C))
begin
    mark_{P_{-1}} = φ;
    if n = 0,
        mark_{P_0} = get_conditioned_slice(P, 0, ⟨a, i, V_h⟩)
    else if n > 0
        for every time step t ≥ 0, while t ≤ n
        begin
            mark_{P_t} = mark_{P_{t-1}} ∐ get_conditioned_slice(P, t, ⟨a, i, V_h⟩)
        end
    mark_P = mark_{P_n}
    S_a = prune(mark_P)
    S_a = get_static_slice (S_a, i, V_H)
    return (S_a)
end
compute_fixpoint_expression (P, a)
begin
    E_{(k+1)} = T
    E_k = (antecedent)
    while ( E_k != E_{k+1} ) do
    begin
        for each process p in P do
            for each statement v = f(X) in process p
                if ( v is a variable in E_k )
                begin
                    sym_v = symbolic_expression(v = f(X), (k - 1))
                    E_{k-1} = E_{k-1} ∨ sym_v
                end
        E_k = E_k ∨ E_{k-1}
        k = k - 1
    end
    return (E_k)
end
get_conditioned_slice (P, t, < a, e, V_h >)
begin
    E_k = compute_fixpoint_expression (P, a)
    for every process p in P do
    begin
        for every statement s in process p do
        begin
            sym_s = symbolic_expression(s, k + t)
            mark_s = decision_procedure ( sym_s , E_k)
        end
    end
    return mark_P
end
prune(P)
begin
    S = P
    for every path ρ in P
        for every sub-path ρ' in ρ until e
            if every statement s in ρ' has (mark_s == F)
                S = S - ρ'
    return (S)
end
```

Figure 2. Algorithm for antecedent conditioned slicing.

The procedure $antecedent_conditioned_slice()$ takes a Verilog program and an LTL property h as input. If the property is of the type $a \Rightarrow X^{=n}c$, where $n = 0$ and a and c are propositional formulas, the $get_conditioned_slice()$ procedure is called. The $get_conditioned_slice()$ procedure returns a *marked* Verilog program. The marked program is now pruned. If the property is of the form $a \Rightarrow X^{n}c$, where $n > 0$, the program is "unrolled" n steps into the future. The $get_conditioned_slice()$ procedure is called repeatedly, for every successive "unroll" of the program into the future. Each unroll is annotated with the corresponding value of the time step t. The union (denoted by the symbol $\coprod$) of marked programs returned by every call to $get_conditioned_slice()$ gives the desired marked program. (The $\coprod$ of two marked programs is defined at every statement as $T \coprod T = T$, $F \coprod F = F$, $T \coprod F = T$, $T \coprod X = T$, $F \coprod X = X$). The final marked program is pruned. 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], [26].

The procedure $get_conditioned_slice()$ takes as input, a Verilog program, the time step t, and the slicing criterion, $\langle a, i, V_h \rangle$, where a is the antecedent, i is the program point where the property is declared and V_h are the variables in the property h. It returns a marked program as an output. The procedure $compute_fixpoint_expression()$ is called. This procedure returns the symbolic expression for the antecedent in the "current" (k^{th}) cycle after unrolling the program. The $get_conditioned_slice()$ procedure at any time t marks the program graph with the truth values of the antecedent at time t. In the case where the property is of the type $a \Rightarrow X^{=n}c$, where $n = 0$, the procedure is called with $t = 0$. The marking procedure is called over all the processes

in the program. For every statement, the symbolic expression of the statement is compared to the symbolic expression of the antecedent using a decision procedure like a rewriting engine or a SAT solver. The procedure can return T, F or X. A statement is marked T in the slice if it is executed t time steps later from the time step k when the antecedent a holds true. A statement is marked F if it is executed t time steps later from the time step k when a is false. If the statement does not, in any future, depend on the truth value of a, it is marked X in the slice. In the case where the decision procedure, typically, a rewriting engine, is not able to determine the truth of the antecedent at a statement, it would return an X.

The procedure $compute_fixpoint_expression()$ takes a Verilog program and the antecedent as input and gives the symbolic expression E_k corresponding to the antecedent as an output. It performs an "unrolling" to eliminate the loops in the *always* blocks of a Verilog program. The antecedent is assumed to be true in the current time cycle k. In the first backward unroll, the program statements that, if executed in the previous ($(k-1)^{th}$) cycle, would cause the antecedent to be true in the current cycle, are isolated. Similarly, in successive unrolls, all the statements in previous cycles that would cause the antecedent to be true in the current cycle are isolated. For each of these statements, the symbolic expression is computed using the $symbolic_expression()$ function. The symbolic expression of a statement s at time k is a Boolean expression that contains the guards as well as the assignments that are required to execute the statement at time k.

The expression E_k contains the symbolic value of the antecedent in the first step. At the end of a single unroll (iteration in the algorithm), the symbolic expressions of all the statements that define the variables in E_k are computed. These are composed in E_{k-1} to obtain the new expression for the next iteration. When the symbolic expressions start repeating over iterations, a fixpoint is reached. This would mean that all possible execution paths that can cause the antecedent to be true, have

been included in the final expression, E_k.

The $prune()$ procedure takes a marked program as input and returns a slice S as an output. It inspects every path in the program control flow graph. For any segment on a path, if there is an F on every statement (node in the control flow graph) in the path, until the endpoint e (leaf node) of a process, it slices the path. Otherwise, it retains the path.

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 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 the instruction being

verified[1]. 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 subsequent 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.

[1]We capture the desired behavior of the antecedent by iteratively refining it using the counterexample generated by a model checker.

[2]We also allow liveness properties in our property specification language. However, verification related properties are typically safety properties.

We outline the algorithm for verifying an instruction I of processor P using our technique in Figure 3. We compare it with the generic antecedent slicing algorithm in Figure 2, in order to show why instruction based slicing for processor verification is an excellent application for the antecedent conditioned slicing technique.

Let $h = [G(I \Rightarrow R)]$ be an LTL formula. Let $I = i_1 \wedge Xi_2 \wedge XXi_3 \ldots X^n i_n$ be the antecedent of h, where i_t represents the antecedent in cycle [3] t. R represents the expected result of executing instruction I, in terms of target register values.

Please note that the antecedent in the first cycle is the instruction word. The antecedents in successive cycles are typically control signals whose values need to be specified for every cycle.

The procedure $insn_check()$ computes the antecedent conditioned slice over the specified k time steps. It is similar to the $antecedent_conditioned_slice()$ procedure in Figure 2. 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 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 $\langle C, e, V \rangle$.

[3]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
  $h = [G(I \Rightarrow X^{=n} R)]$
  $insn_check(P, h)$
end
insn_check (P, h)
begin
    $mark_{P_{-1}} = \phi$
    for every time step $t \geq 0$, while $t \leq n$
    begin
        $mark_{P_t} = mark_{P_{t-1}} \coprod get_conditioned_slice(P, t, \langle I, i, V_h \rangle)$
    end
    $mark_P = mark_{P_n}$
    S_a = prune($mark_P$)

    S_a = get_static_slice (S_a, i, V_H)
    return $model_check(S_a, h)$
end
get_conditioned_slice ($P, t, < a, i, V_h >$)
begin
    for every process p in P do
    begin
        for every statement s in process p do
        begin
            sym_s = symbolic_expression(s, t)
            $mark_s$ = decision_procedure (sym_s , a)
        end
    end
    return $mark_P$
end
prune(P)
begin
    $S = P$
    for every path ρ in P
        for every sub-path ρ' in ρ until e
            if for every statement s in the ρ' ($mark_s$ == F)
                $S = P - \rho'$
    return (S)
end

Figure 3. Algorithm for verifying an instruction I using antecedent conditioned slicing.

3.2. Expediency of Applying Antecedent Conditioned Slicing to Processor Verification

When antecedent conditioned slicing is applied to instruction slicing for processors, some features are observed, that are advantageous with respect to efficiency and lower computational effort. The important considerations that provide performance benefits are as follows.

- The antecedent is the instruction (word), which is an input, in the case of processor verification. This eliminates the call to $compute_fixpoint_expression()$, in the $get_conditioned_slice()$ procedure, that was a part of the generic algorithm. Since the antecedent is an input, it is not necessary to find the symbolic expression of the antecedent. This reduces the corresponding computational effort. This is a substantial reduction in the overall computational complexity of the algorithm. In theory, this step can be computationally intensive, and eliminating it increases the efficiency of the technique considerably.

- In instruction based slicing, an instruction's complete behavior, from the stage when it is fetched, to the stage when it is written back, is isolated. This provides an "independence" of every slice from the other parts of the Verilog program, since the other parts represent behaviors of other instructions.

  In order to determine the truth value of the antecedent at a given statement, the $decision_procedure()$ in the generic algorithm could be a rewriting engine or a SAT solver that would evaluate the contradiction between two Boolean expressions. This could be a computationally intensive step that depends heavily on the efficiency of the decision making engine, especially if the two expressions are complicated. However, in the case of instruction-based slicing, the decision making procedure is relatively much simpler. This is due to the independent nature of the system behavior with respect to each instruction that makes the symbolic expressions for

the Verilog program variables more tractable than for a generic Verilog program. Also, since the antecedent is an input, and not assigned within the program, the symbolic expression for the antecedent is not complicated. This makes the evaluation of an instruction's truth value at any point in the program significantly less complex than the generic case. Consequently, the cases where the decision procedure would return an X due to its inability to determine the truth of the antecedent, will be minimal.

- The antecedent (instruction), once it is true, does not change its truth value for the duration of the property checking. The $prune()$ procedure in the instruction slicing algorithm is very simple in the case of instruction slicing. In the generic case, where the truth value of the antecedent is liable to change (toggle) all paths of the program have to be analyzed for toggling behavior. If toggling is observed, the entire path needs to be retained. Since in the case of processor designs, the truth value of the antecedent *does not change along a path*, the path traversal step can be optimized to be more efficient. If the antecedent is not true at a node (program statement) in the control flow graph, it can be assumed that along all subsequent paths from that node until an endpoint e, the antecedent will not be true. Due to this "independence" in system behavior according to an instruction, a large portion of the Verilog program can be sliced away, when considering a single instruction.

We have thus shown how the complexity of the generic antecedent conditioned slicing algorithm is greatly simplified when applied to instruction based slicing of processors. As a result, the algorithm can be implemented very efficiently for this application.

3.3. Correctness of our notion of pipelined processor verification

We present an argument to show that our notion of pipelined processor verification is correct and complete.

Let P be a pipelined processor with n stages in its pipeline. Let $O(P)$ be the observed state of P, consisting of the register file, the memory, the non-programmer-visible (temporary) registers and the program counter at any cycle. $O(P)$ is therefore, a superset of the programmer visible state. Let $I_1, I_2 \dots I_n$ be the instructions at any point of time in the pipeline. If all these instructions execute correctly, they update $O(P)$. Let ψ be the antecedent conditioned slicing operation that performs the instruction-wise decomposition of P. If $Q = \psi(P)$, then Q represents the set of all instruction slices $Q_1, Q_2 \dots Q_k$, where k is the size of the instruction set of P. Let $Q_1, Q_2 \dots Q_n$ be the instruction slices corresponding to the an arbitrary sequence of instructions $I_1, I_2 \dots I_n$.

Let $O(Q_n)$ be the observed state of Q_n. Let ξ be an execution operation of an instruction that updates the observed state of P, after executing a sequence of zero or more instructions. Then, the final observed state after executing instructions $I_1 \dots I_n$ in P's pipeline (shown by $\|$) is given by $O(P_n)$.

$$O(P_n) = \xi(O(P), (I_1 \parallel I_n))$$

If now, $Q_1, Q_2 \dots Q_n$ are executed in sequence, the following operations represent the updated final state of Q.

$$O'(P_1) = \xi(O(P), Q_1)$$

$$O'(P_2) = \xi(O'(P_1), Q_2)$$

$$\vdots$$

$$O'(P_n) = \xi(O(P_{n-1}), Q_{n-1})$$

Then, $O(P_n) = O'(P_n)$.

Proof Intuition

The above theorem states that the instruction slices, when executed in the same sequence as the corresponding instructions in the original pipelined machine, will produce the same state as the original pipeline. In order for this theorem to be true, we need to prove the following.

- Each instruction slice $Q_1 \ldots Q_n$ executes correctly, *i.e* it *produces the same result* as $I_1 \ldots I_n$ respectively, with respect to updating the target registers.
- Each instruction slice $Q_1 \ldots Q_n$ *does not alter* the processor state incorrectly at any cycle during its execution, *i.e* the visible and non-visible registers do not get updated with a wrong value.

These two steps provide the necessary and sufficient conditions for the correctness of the pipelined processor using our technique. Therefore, for any given pipelined processor, we need to prove these two results.

4. Verification of the OR1200 processor

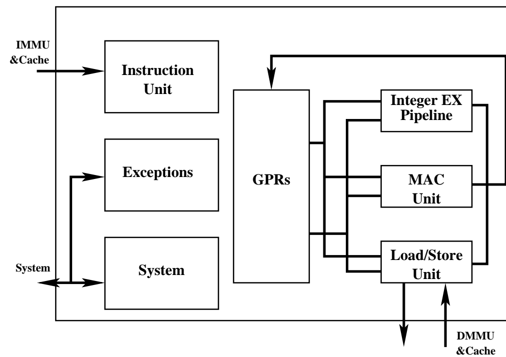

Figure 4. CPU block diagram of OR1200

As discussed in Subsection 3.3 we need to prove two results for proving the correctness of a pipelined processor. 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). 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 [18]. 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. Figure 4 shows the block diagram of the CPU of the OR1200 processor.

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[4] 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 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 conditions:

```
if (!rst && !flushpipe && !no_more_dslot &&
```

[4]This refers to the specification manual of the processor, where each instruction's output values in target registers is given.

```
    !rfe && if_freeze)
```

These two conditions 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 control conditions, as indicated by the counterexample, need to be specified every cycle. For instance, negating the signal `id_freeze` disables the freezing of the decode 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 and performed 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 lemma1:
      ((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 to 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 [17] with the abstraction refinement (absref) 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.

In Table 2, we present the results using different options in the SMV model checker. Each option corresponds to the implementation of a different algorithm. In order to analyze the efficacy and efficiency of our static analysis technique with different verification algorithms, we ran the antecedent conditioned slices using various Boolean level engines.

The column Localization shows the results of running a counterexample based localization refinement engine. Localization reduction with counterexample guided refinement was introduced by Kurshan in [14]. Localization reduction is an iterative technique that starts with an abstraction of the model under verification and tries to verify the property. When a counterexample is found, a reconstruction process is executed to determine if it is a valid one. If the counterexample is found to be spurious, the abstract model is refined to eliminate the possibility of this counterexample in the next iteration. This algorithm uses BDD-based lower level engines. These results have been

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
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. Time 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).

Class	Instruction	Localization	k-induction
ALU	l.addc	25.14s	16.71s
ALU	l.addic	17.84s	13.16s
BRANCH	l.j	57.36s	427.07s
BRANCH	l.jal	54.98s	396.73s
MAC	l.mul	25.28s	19.72s
MAC	l.mulu	26.63s	21.05s
COMPARE	l.sfne	183.01s	157.50s
COMPARE	l.sfgt	194.43s	179.81s
LSU	l.ld	35.85s	32.70s
LSU	l.sw	39.30s	37.00s
SHF/ROT	l.srl	27.83s	17.40s
SHF/ROT	l.sra	28.42s	23.26s

Table 2. Time taken in seconds after antecedent conditioned slicing, by engines implementing different Boolean level algorithms in SMV. The results are shown for two instructions per instruction class.

Instruction Class	Instruction	STE Unsliced	STE Sliced
ALU	l.addc	549.12s	431.37s
ALU	l.addic	551.53s	459.41s
BRANCH	l.j	811.87s	539.90s
BRANCH	l.jal	804.41s	781.61s
MAC	l.mul	570.92s	420.32s
MAC	l.mulu	575.83s	395.54s
COMPARE	l.sfne	905.61s	641.59s
COMPARE	l.sfgt	899.75s	705.81s
LSU	l.ld	834.00s	597.42s
LSU	l.sw	800.50s	522.19s
SHF/ROT	l.srl	795.61s	505.89s
SHF/ROT	l.sra	801.37s	526.23s

Table 3. Time taken by an STE engine to verify the sliced and unsliced versions of the design

obtained for runs without sifting or ordering the BDDs in these engines. The column k-induction corresponds to a safety property checking algorithm based on induction with depth, strengthened with a constraint that all states in a path be unique. This algorithm uses a SAT solver as a lower level engine, as described in [23].

The original processor model, without preprocessing it with our technique, does not run to completion with any of these engines due to capacity issues.

We also used our static analysis technique to decompose the design before passing it through a Symbolic Trajectory Evaluation(STE) [5] engine as shown in Table 3. The STE engine completed the verification of the unsliced version of the design. The table shows that the benefits obtained over the STE algorithm are not as significant as over the other verification algorithms. This could be due to the similarity between the constraint based reduction heuristics used by the tool and our slicing algorithm. However, considerable manual effort was spent in specifying the input constraints of the design for the STE engine.

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 [21] 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 [27], [28] 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], [20], [22]. 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 Boolean level techniques. Consequently, only isolated components like the floating point execution unit, or the instruction decode logic are verified using these techniques [4], since the state of the art tools cannot scale to the entire processor design.

Patankar et al [19] 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 is related to ours, since it is oriented toward processor verification on an instruction by instruction basis, as opposed to the Burch and Dill correctness criterion. However, their technique requires significant manual intervention to decompose the proof into manageable

pieces using symmetry, temporal case splitting and abstract interpretation. Our abstractions, 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

We have introduced an abstraction technique that can make model checking of processors viable. The abstraction exploits the natural decomposition that instructions in a pipelined processor allow. Our technique can be viewed as an RT-level static analysis step that can be built on lower level Boolean engines, to increase their efficiency and scope. In its current form, our technique is most effective for single instruction issue, multi-stage pipeline processors, such as graphics and embedded processors.

Since we use abstractions created by syntactic transformations, the efficiency of our algorithm is program and property dependent. In the case of properties where the antecedent changes over a period, all future behavior of the program would need to be retained for each time step. In these cases, the antecedent conditioned slice would be quite large.

In that respect, processor verification is an excellent application for our abstractions. The antecedent, which is the instruction word in the single instruction machine, does not change through the duration of the property.

In order to specify the correct antecedent, we use the counterexample from a model checker to

refine the antecedent. This could potentially be manually intensive. In future work, we plan to automate this step.

Our future work involves applying the technique to processors with more complex features, like out-of-order execution, register renaming, interrupts etc.

References

[1] OpenRISC 1200 Properties. *http://www.cerc.utexas.edu/∼shobha/OR1200_properties/*.

[2] M. Aagaard, V. Ciubotariu, 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.

[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*, 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] R. Kurshan. *Computer-Aided-Verification of Coordinating Processes*. Princeton University Press, 1994.

[15] 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.

[16] 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.

[17] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37(1-3):279–309, 2000.

[18] OPENCORES. http://www.opencores.org.

[19] 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.

[20] 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.

[21] 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.

[22] T. Schubert. High level formal verification of next-generation microprocessors. In *Proceedings of the 40th conference on Design automation*, pages 1–6, 2003.

[23] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *FMCAD*, pages 108–125, 2000.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] M. N. Velev and R. E. Bryant. Formal verification of superscale microprocessors with multicycle functional units, exception, and branch prediction. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 112–117, 2000.

[29] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, 10(4):352–357, 1984.