Design Automation for Embedded Systems, 8, 249–265, 2003.
© 2003 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

# Towards Automatic Validation of Dynamic Behavior in Pipelined Processor Specifications

PRABHAT MISHRA pmishra@cecs.uci.edu
*Center for Embedded Computer Systems, University of California, Irvine 92697 CA, USA*

NIKIL DUTT dutt@cecs.uci.edu
*Center for Embedded Computer Systems, University of California, Irvine 92697 CA, USA*

HIROYUKI TOMIYAMA tomiyama@is.nagoya-u.ac.jp
*Department of Information Engineering, Nagoya University, Furo-cho, Chikusa-ku, Nagoya, 464-8603, Japan*

**Abstract.** As embedded systems continue to face increasingly higher performance requirements, deeply pipelined processor architectures are being employed to meet desired system performance. A significant bottleneck in the validation of such systems is the lack of a golden reference model. Thus, many existing techniques employ a bottom-up approach to architecture validation, where the functionality of an existing pipelined architecture is, in essence, reverse-engineered from its implementation. Our validation technique is complementary to these bottom-up approaches. Our approach leverages the system architect's knowledge about the behavior of the pipelined architecture, through Architecture Description Language (ADL) constructs, and thus allows a powerful top–down approach to architecture validation. The most important requirement in top–down validation process is to ensure that the specification (reference model) is golden. Earlier, we have developed validation techniques to ensure that the static behavior of the pipeline is well-formed by analyzing the structural aspects of the specification using a graph based model. In this paper, we verify the dynamic behavior by analyzing the instruction flow in the pipeline using a Finite State Machine (FSM) based model to validate several important architectural properties such as determinism and in-order execution in the presence of hazards and multiple exceptions. We applied this methodology to the specification of a representative pipelined processor to demonstrate the usefulness of our approach.

**Keywords:** Architecture specification, determinism, in-order execution, pipeline validation.

## 1. Introduction

Embedded systems present a tremendous opportunity to customize designs by exploiting the application behavior. Shrinking time-to-market, coupled with short product lifetimes create a critical need for rapid exploration and evaluation of candidate System-on-Chip (SoC) architectures. System architects critically need tools, techniques and methodologies to perform rapid architectural exploration for a given set of applications to meet diverse requirements, such as better performance, low power, smaller area, and improved clock frequency. Recent approaches use Architecture Description Languages (ADL) to specify the architecture. The software toolkit including compiler and simulator is generated automatically from this ADL specification to enable rapid design space exploration.

The ADL driven design space exploration has been addressed extensively in both academia ([1], [4], [12], [21], [25]) and industry [24]. However, the validation of the

ADL specification has not been addressed so far. It is important to validate the ADL description of the architecture to ensure the correctness of both the architecture specified, as well as the generated software toolkit. The benefits of validation are twofold. First, the process of any specification is error-prone and thus verification techniques can be used to check for correctness and consistency of specification. Second, changes made to the processor during exploration may result in incorrect execution of the system and verification techniques can be used to ensure correctness of the modified architecture.

Furthermore, the validated ADL specification can be used as a golden reference model for processor pipeline validation. One of the most important problems in today's processor design validation is the lack of a golden reference model that can be used for verifying the design at different levels of abstraction. Thus many existing validation techniques employ a bottom-up approach to pipeline verification, where the functionality of an existing pipelined processor is, in essence, reverse-engineered from its RT-level implementation. Our validation technique is complementary to these bottom up approaches. Our approach leverages the system architects knowledge about the behavior of the pipelined processor, through ADL constructs, and thus allows a powerful top–down approach to pipeline validation.

In our ADL driven exploration flow, the designer describes the processor architecture in EXPRESSION ADL [1]. To ensure that the ADL specification describes a well-formed architecture we verify both static and dynamic behavior of the processor specification. Earlier, we have developed validation techniques to ensure that the static behavior of the pipeline is well-formed by analyzing the structural aspects of the specification using a graph based model [19]. In this paper, we verify the dynamic behavior of the processor's description by analyzing the instruction flow in the pipeline using a Finite State Machine (FSM) based model to validate several important architectural properties such as determinism and in-order execution in the presence of hazards and multiple exceptions. Our automatic property checking framework determines if all the necessary properties are satisfied or not. In case of a failure, it generates traces so that designer can modify the ADL specification of the architecture. We applied this methodology to a single-issue DLX processor to demonstrate the usefulness of our approach.

The rest of the paper is organized as follows. Section 2 presents related work addressing validation of pipelined processors. Section 3 outlines our validation approach and the overall flow of our environment. Section 4 presents our FSM based modeling of processor pipelines. Section 5 proposes our validation technique followed by a case study in Section 6. Finally, Section 7 concludes the paper.

## 2. Related Work

Several approaches for formal or semi-formal verification of pipelined processors have been developed in the past. Theorem proving techniques, for example, have been successfully adapted to verify pipelined processors ([3], [9], [13]). Hosabettu [22] proposed an approach to decompose and incrementally build the proof of

correctness of pipelined microprocessors by constructing the abstraction function using completion functions.

Burch and Dill presented a technique for formally verifying pipelined processor control circuitry [5]. Their technique verifies the correctness of the implementation model of a pipelined processor against its Instruction-Set Architecture (ISA) model based on quantifier-free logic of equality with uninterpreted functions. The technique has been extended to handle more complex pipelined architectures by several researchers [10], [14].

Huggins and Campenhout verified the ARM2 pipelined processor using Abstract State Machine [6]. In [8], Levitt and Olukotun presented a verification technique, called unpipelining, which repeatedly merges last two pipeline stages into one single stage, resulting in a sequential version of the processor. A framework for microprocessor correctness statements about safety that is independent of implementation representation and verification approach is presented in [11].

Ho et al. [15] extract controlled token nets from a logic design to perform efficient model checking. Jacobi [2] used a methodology to verify out-of-order pipelines by combining model checking for the verification of the pipeline control, and theorem proving for the verification of the pipeline functionality. Compositional model checking is used to verify a processor microarchitecture containing most of the features of a modern microprocessor [20].

All the techniques mentioned above attempt to formally verify the implementation of pipelined processors by comparing the pipelined implementation with its sequential (ISA) specification model, or by deriving the sequential model from the implementation. Our validation technique is complementary to these formal approaches. We define a set of properties which have to be satisfied for the correct pipeline behavior, and verify the correctness of pipelined processor specifications by applying these properties using a FSM-based model.

## 3. Our Validation Approach

Figure 1 shows our ADL driven exploration flow. The designer describes the processor architecture in EXPRESSION ADL [1]. It is necessary to validate the ADL specification to ensure the correctness of the generated software toolkit and the HDL implementation. To ensure that the ADL specification describes a well-formed architecture we verify both static and dynamic behavior of the processor specification.

We have developed validation techniques to ensure that the static behavior of the pipeline is well-formed by analyzing the structural aspects of the specification using several properties, such as connectedness of components, false pipeline and data-transfer paths, operation completeness, and finiteness of execution. These properties are applied on the graph model of the processor specification to verify the static behavior of the specification [19].

We verify the dynamic behavior of the processor by analyzing the instruction flow in the pipeline using a FSM-based model to validate several important architectural

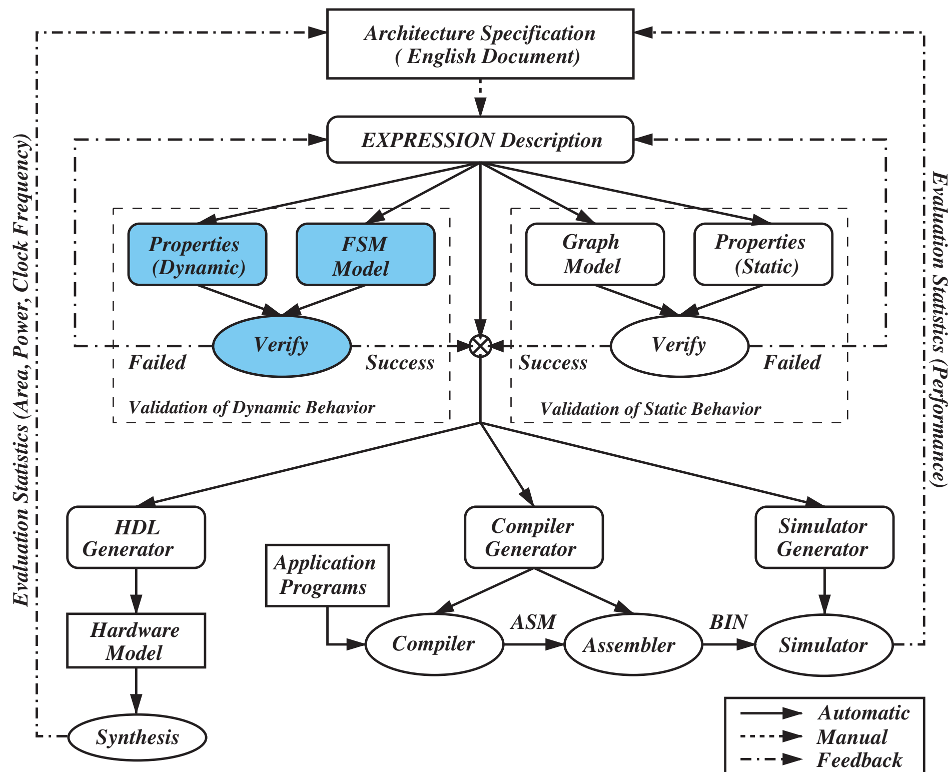


*Figure 1.* ADL driven validation and exploration flow.

properties such as determinism and in-order execution in the presence of hazards and multiple exceptions.

Our automatic property checking framework determines if all the necessary properties are satisfied or not. In case of a failure, it generates traces so that designer can modify the ADL specification of the architecture. If the verification is successful, the software toolkit (including compiler and simulator), and the implementation (hardware model) can be generated for design space exploration. The application program is compiled and simulated to generate performance numbers. The information regarding silicon area, clock frequency, or power consumption is determined by synthesizing the hardware model. The feedback is used by the designer to make modifications to the specification to obtain the best architecture possible for the given set of application programs and design constraints.

## 4. Modeling of Processor Pipelines

In this section we describe how we derive the FSM model of the pipeline from the ADL description of the processor. We first explain how we specify the information

necessary for FSM modeling, then we present the FSM model of the processor pipelines using the information captured in the ADL.

### *4.1. Processor Pipeline Description in ADL*

We have chosen the EXPRESSION ADL [1] that captures the structure and behavior of the processor pipeline. The techniques, we developed, are applicable to any ADL that captures both the structure and the behavior of the architecture.

The structure (of a processor) is defined by its components (units, storages, ports, connections) and the connectivity (pipeline and data-transfer paths) between these components. Each component is defined by its attributes: the list of opcodes it supports, execution timing for each supported opcode etc. The behavior of a processor is defined by its instruction set. Each operation in the instruction-set is defined in terms of opcode, operands and the functionality of the operation. Figure 2(a) shows a fragment of a processor pipeline. The oval boxes represent units, rectangular boxes represent pipeline latches, and arrows represent pipeline edges. In this section we briefly describe how we specify pipeline flow conditions for stalling, normal flow, bubble insertion, exception, and squashing in ADL. The detailed description of how to specify structure, behavior, and pipeline flow conditions in ADL is available in [16].

A unit is in *normal flow* (NF) if it can receive instruction from its parent unit and can send to its child unit. A unit can be *stalled* (ST) due to external signals or due to conditions arising inside the processor pipeline. For example, the external signal that can stall a fetch unit is *ICache_Miss*; the internal conditions to stall the fetch unit can

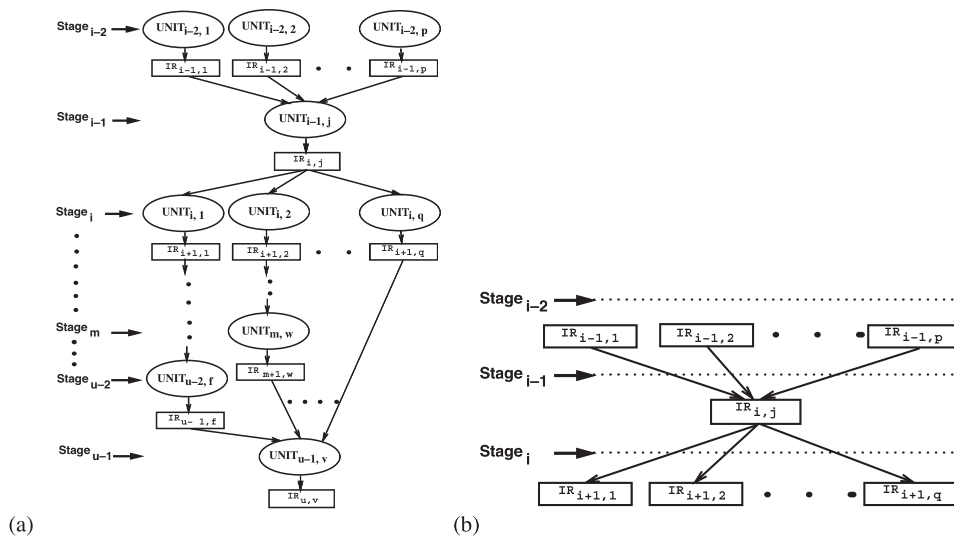


*Figure 2.* A fragment of the processor pipeline. (a) With units and pipeline latches. (b) With only instruction registers.

be due to decode stall, hazards, or exceptions. A unit performs *bubble insertion* (BI) when it does not receive any instruction from its parent (or busy computing in case of multicycle unit), and its child unit is not stalled. A unit can be in *exception* condition due to internal contribution or due to an exception. A unit is in bubble/nop *squashed* (SQ) stage when it has nop instruction that gets removed or overwritten by an instruction of the parent unit.

For units, with multiple children the flow conditions due to internal contribution may differ. For example, the unit $UNIT_{i-1,j}$ in Figure 2(a) with $q$ children can be *stalled* when *any* one of its children is stalled, or when *some* of its children are stalled (designer identifies the specific ones), or when *all* of its children are stalled; or when *none* of its children are stalled. During specification, the designer selects from the set ($ANY$, $SOME$, $ALL$, $NONE$) the internal contribution along with any external signals to specify the stall condition for each unit. Similarly, the designer specifies the internal contribution for other flow conditions [16].

The PC unit can be *stalled* (ST) due to external signals such as cache miss or when the fetch unit is stalled. When a branch is taken the PC unit is said to be in *branch taken* (BT) state. The PC unit is in *sequential execution* (SE) mode when the fetch unit is in normal flow, there are no external interrupts, and the current instruction is not a branch instruction.

### *4.2. FSM Model of Processor Pipelines*

This section presents an FSM-based modeling of controllers in pipelined processors. Figure 2(b) shows a fragment of the processor pipeline with only instruction registers. We assume a pipelined processor with in-order execution as the target for modeling and validation. The pipeline consists of $n$ stages. Each stage can have more than one pipeline registers (in case of fragmented pipelines). Each single-cycle pipeline register takes one cycle if there are no pipeline hazards. A multi-cycle pipeline register takes $m$ cycles during normal execution (no hazards). In this paper we call these pipeline registers *instruction registers* (IR) since they are used to transfer instructions from one pipeline stage to the next. Let $Stage_i$ denote the $i$th stage where $0 \leq i \leq n-1$, and $n_i$ the number of pipeline registers between $Stage_{i-1}$ and $Stage_i$. Let $IR_{i,j}$ denote an instruction register between $Stage_{i-1}$ and $Stage_i$ ($1 \leq i \leq n$, $1 \leq j \leq n_i$). The first stage, i.e., $Stage_0$, fetches an instruction from instruction memory pointed by program counter $PC$, and stores the instruction into the first instruction register $IR_{1,j}$ ($1 \leq j \leq n_1$). Without loss of generality, let us assume that $IR_{i,j}$ has $p$ parent units and $q$ *children* units as shown in Figure 2(b). During execution the instruction stored in $IR_{i,j}$ is executed at $Stage_i$ and then stored into the next instruction register $IR_{i+1,k}$ ($1 \leq k \leq q$). In this paper, we define a state of the $n$-stage pipeline as values of $PC$ and $\sum_{i=1}^{n-1} n_i$ instruction registers, where $n_i$ is the number of pipeline registers between $Stage_{i-1}$ and $Stage_i$ ($1 \leq i \leq n$). Let $PC(t)$ and $IR_{i,j}(t)$ denote the values of $PC$ and $IR_{i,j}$ at time $t$, respectively. Then, the state of the pipeline at time $t$ is defined as

$$S(t) = \langle PC(t), IR_{1,1}(t), \ldots, IR_{i,j}(t), \ldots, IR_{n-1}, n_{n-1}(t) \rangle \tag{1}$$

We first describe the flow conditions for stalling (ST), normal flow (NF), bubble insertion (BI), bubble squashing (SQ), sequential execution (SE), and branch taken (BT) in the FSM model, then we describe the state transition functions possible in the FSM model using the flow conditions.

In this paper we use "$\vee$" to denote logical *or*, and "$\wedge$" to denote logical *and*. For example, $(a \vee b)$ implies ($a$ or $b$), and $(a \wedge b)$ implies ($a$ and $b$). We use $\bigvee_i^j$ and $\bigwedge_i^j$ to denote sum and product of symbols respectively. For example, $\bigvee_{i=0}^2 a_i$ implies $(a_0 \vee a_1 \vee a_2)$, and $\bigwedge_{i=0}^2 a_i$ implies $(a_0 \wedge a_1 \wedge a_2)$.

### 4.2.1. *Modeling conditions in FSM*

Let us assume, every instruction register $IR_{i,j}$ has an exception bit $XN_{IR_{i,j}}$, which is set when the exception condition ($cond_{IR_{i,j}}^{XN}$ say) is true. The $XN_{IR_{i,j}}$ has two components viz., exception condition when the children are in exception ($XN_{IR_{i,j}}^{child}$ say) and exception condition due to exceptions on $IR_{i,j}$ ($XN_{IR_{i,j}}^{self}$ say). More formally the exception condition at time $t$ in the presence of a set of external signals $I(t)$ on $S(t)$ is, $cond_{IR_{i,j}}^{XN}(S(t), I(t))$ ($cond_{IR_{i,j}}^{XN}$ in short),

$$cond_{IR_{i,j}}^{XN} = XN_{IR_{i,j}} = XN_{IR_{i,j}}^{child} \vee XN_{IR_{i,j}}^{self} \tag{2}$$

For example, if designer specified that "ANY" (see Section 4.1) of the children are responsible for the exception on $IR_{i,j}$ i.e., $IR_{i,j}$ will be in exception condition if any of its children is in exception, the equation 2 becomes:

$$XN_{IR_{i,j}} = \left( \bigvee_{k=1}^{q} XN_{IR_{i+1,k}} \right) \vee XN_{IR_{i,j}}^{self}$$

Similarly, the conditions for squashing ($cond_{IR_{i,j}}^{SQ}$ say), stalling ($cond_{IR_{i,j}}^{ST}$ say), normal flow ($cond_{IR_{i,j}}^{NF}$ say) and bubble insertion ($cond_{IR_{i,j}}^{BI}$ say) are shown below.

$$cond_{IR_{i,j}}^{SQ} = SQ_{IR_{i,j}} = NF_{IR_{i,j}}^{parent} \wedge ST_{IR_{i,j}}^{child} \wedge ((IR_{i,j}).opcode == nop) \tag{3}$$

$$cond_{IR_{i,j}}^{ST} = (ST_{IR_{i,j}}^{child} \vee ST_{IR_{i,j}}^{self}) \wedge \overline{XN_{IR_{i,j}}} \wedge \overline{SQ_{IR_{i,j}}} \tag{4}$$

$$cond_{IR_{i,j}}^{NF} = NF_{IR_{i,j}}^{parent} \wedge NF_{IR_{i,j}}^{child} \wedge \overline{ST_{IR_{i,j}}^{self}} \wedge \overline{XN_{IR_{i,j}}} \wedge \overline{SQ_{IR_{i,j}}} \tag{5}$$

$$cond_{IR_{i,j}}^{BI} = BI_{IR_{i,j}}^{parent} \wedge BI_{IR_{i,j}}^{child} \wedge \overline{ST_{IR_{i,j}}^{self}} \wedge \overline{XN_{IR_{i,j}}} \wedge \overline{SQ_{IR_{i,j}}}. \tag{6}$$

Similarly the conditions for PC viz., $cond_{PC}^{SE}$ (SE: sequential execution), $cond_{PC}^{BI}$ (BI: bubble insertion), and $cond_{PC}^{BT}$ (BT: branch taken) can be described using the information available in the ADL. The $cond_{PC}^{BT}$ is true when a branch is taken or

when an exception is taken. When a branch is taken, the PC is modified with the target address. When an exception is taken, the PC is updated with the corresponding interrupt service routine address. Let us assume, $BT_{PC}$ bit is set when the unit completes execution of a branch instruction and the branch is taken. Formally,

$$cond_{PC}^{SE}(S(t), I(t)) = NF_{PC}^{child} \wedge \overline{ST_{PC}^{self}} \wedge \overline{BT_{PC}} \wedge \overline{XN_{IR_{1,j}}} \tag{7}$$

$$cond_{PC}^{ST}(S(t), I(t)) = (ST_{PC}^{child} \vee ST_{PC}^{self}) \wedge \overline{BT_{PC}} \wedge \overline{XN_{IR_{1,j}}} \tag{8}$$

$$cond_{PC}^{BT}(S(t), I(t)) = (BT_{PC} \vee XN_{IR_{1,j}}) \tag{9}$$

### 4.2.2. *Modeling State Transition Functions*

In this section, we describe the next-state function of the FSM. Figure 2(b) shows a fragment of the processor pipeline with only instruction registers. If there are no pipeline hazards, instructions flow from IR (instruction register) to IR every $m$ cycles ($m = 1$ for single-cycle IR). In this case, the instruction in $IR_{i-1,l}$ ($1 \le l \le p$) at time $t$ proceeds to $IR_{i,j}$ after $m$ cycles ($m$ is the *timing* of $IR_{i-1,1}$ and $IR_{i,j}$ has $p$ parent latches and $q$ child latches as shown in Figure 2(b)), i.e., $IR_{i,j}(t+1) = IR_{i-1,l}(t)$. In the presence of pipeline hazards, however, the instruction in $IR_{i,j}$ may be stalled, i.e., $IR_{i,j}(t+1) = IR_{i,j}(t)$. Note that, in general, any instruction in the pipeline cannot skip pipeline stages. For example, $IR_{i,j}(t+1)$ cannot be $IR_{i-2,v}(t)$ ($1 \le v \le n_{i-2}$) if there are no feed-forward paths.

The rest of this section formally describes the next-state function of the FSM. According to the equation 1, a state of an $n$-stage pipeline is defined by $(M+1)$ registers (PC and $M$ instruction registers where, $M = \sum_{i=1}^{n-1} n_i$). Therefore, the next state function of the pipeline can also be decomposed into $(M+1)$ sub-functions each of which is dedicated to a specific state register. Let $f_{PC}^{NS}$ and $f_{IR_{i,j}}^{NS}$ ($1 \le i \le n-1$, $1 \le j \le n_i$) denote next-state functions for $PC$ and $IR_{i,j}$, respectively. Note that in general $f_{IR_{i,j}}^{NS}$ is a function of not only $IR_{i,j}$ but also other state registers and external signals from outside of the controller. For program counter, we define three types of state transitions as follows.

$$PC(t+1) = f_{PC}^{NS}(S(t), I(t)) = \begin{cases} PC(t) + L & \text{if } cond_{PC}^{SE}(S(t), I(t)) = 1 \\ target & \text{if } cond_{PC}^{BT}(S(t), I(t)) = 1 \\ PC(t) & \text{if } cond_{PC}^{ST}(S(t), I(t)) = 1 \end{cases} \tag{10}$$

Here, $I(t)$ represents a set of external signals at time $t$, $L$ represents the instruction length, and *target* represents the branch target address which is computed at a certain pipeline stage. The $cond_{PC}^{x}$'s ($x \in SE$, $BT$, $ST$) are logic functions of $S(t)$ and $I(t)$ as described in equations 7–9, and return either 0 or 1. For example, if $cond_{PC}^{ST}(S(t), I(t))$ is 1, $PC$ keeps its current value at the next cycle.

For instruction registers, $IR_{i,j}$ $(2 \leq i \leq n-1,\ 1 \leq j \leq n_i)$, we define five types of state transitions as follows. The state transitions for the first instruction register, $IR_{1,j}$, will have $IM(PC(t))$ in place of $IR_{i-1,l}(t)$, where $IM(PC(t))$ denotes the instruction pointed by the program counter (PC) in instruction memory (IM).

$$IR_{i,j}(t+1) = f_{i,j}^{NS}(S(t), I(t)) = \begin{cases} IR_{i-1,l}(t) & \text{if } cond_{IR_{i,j}}^{NF}(S(t), I(t)) = 1 \\ IR_{i,j}(t) & \text{if } cond_{IR_{i,j}}^{ST}(S(t), I(t)) = 1 \\ nop & \text{if } cond_{IR_{i,j}}^{BI}(S(t), I(t)) = 1 \\ IR_{i-1,l}(t) & \text{if } cond_{IR_{i,j}}^{SQ}(S(t), I(t)) = 1 \\ nop & \text{if } cond_{IR_{i,j}}^{XN}(S(t), I(t)) = 1 \end{cases} \quad (11)$$

The $IR_{i,j}$ is said to be stalled at time $t$ if $cond_{IR_{i,j}}^{ST}(S(t), I(t))$ is 1, resulting in $IR_{i,j}(t+1) = IR_{i,j}(t)$. Similarly, $IR_{i,j}$ is said to flow normally at time $t$ if $cond_{IR_{i,j}}^{NF}(S(t), I(t))$ is 1. A *nop* instruction (bubble) is inserted in $IR_{i,j}$ when $cond_{IR_{i,j}}^{BI}(S(t), I(t))$ or $cond_{IR_{i,j}}^{XN}(S(t), I(t))$ is 1, resulting in $IR_{i,j}(t+1) = nop$. Similarly, when $cond_{IR_{i,j}}^{SQ}(S(t), I(t))$ is 1, the bubble in $IR_{i,j}$ gets overwritten by the instruction from the parent instruction register, i.e., $IR_{i,j}(t+1) = IR_{i-1,l}(t)$ $(1 \leq l \leq n_{i-1})$.

At present, signals coming from the datapath or the memory subsystem into the pipeline controller are modeled as primary inputs to the FSM, and control signals to the datapath or the memory subsystem are modeled as outputs from the FSM.

## 5. Validation of Processor Specification

Based on the FSM modeling presented in Section 4, we propose a method for validating pipelined processor specifications using two properties: determinism and in-order execution.We first describe the properties needed for validating the specification, then we present an automatic property checking framework driven by EXPRESSION ADL [1].

### 5.1. Properties

This section presents two properties: determinism and in-order execution. Any pipelined processor with in-order execution must satisfy these properties.

#### 5.1.1. Determinism

To ensure correct execution, there should not be any instruction or data loss in the pipeline. The bubble squashing and flushing of instructions are permitted. The flushed instructions are fetched and executed again. The next-state functions for all

state registers must be deterministic. This property is valid if all the following equations hold for $\forall i,j$ $(1 \leq i \leq n-1,\ 1 \leq j \leq n_i)$.

$$cond_{PC}^{SE} \vee cond_{PC}^{BT} \vee cond_{PC}^{ST} = 1 \tag{12}$$

$$cond_{IR_{i,j}}^{NF} \vee cond_{IR_{i,j}}^{ST} \vee cond_{IR_{i,j}}^{BI} \vee cond_{IR_{i,j}}^{XN} \vee cond_{IR_{i,j}}^{SQ} = 1 \tag{13}$$

$$\forall x, y(x, y \in \{SE, BT, ST\} \wedge x \neq y), \quad cond_{PC}^{x} \wedge cond_{PC}^{y} = 0 \tag{14}$$

$$\forall x, y(x, y \in \{NF, ST, BI, XN, SQ\} \wedge x \neq y), \quad cond_{IR_{i,j}}^{x} \wedge cond_{IR_{i,j}}^{y} = 0 \tag{15}$$

The first two equations mean that, in the next-state function for each state register, the five conditions must cover all possible combinations of processor states $S(t)$ and external signals $I(t)$. The last two equations guarantee that any two conditions are disjoint for each next-state function. Informally, exactly one of the conditions should be true in a clock cycle for each state register. As a result, at any time $t$ an instruction register will have a deterministic instruction.

### *5.1.2. In-order Execution*

A pipelined processor with in-order execution is correct if all instructions which are fetched from instruction memory, flow from the first stage to the last stage, while maintaining their execution order. In order to guarantee in-order execution, state transitions of adjacent instruction registers must depend on each other. Illegal combination of state transitions of adjacent stages are described below using Figure 2 where $2 \leq i \leq n-1, 1 \leq j \leq n_i, 1 \leq l \leq p$, and $1 \leq k \leq q$.

An instruction register can not be in normal flow if all the parent instruction registers (adjacent ones) are stalled. If such a combination of state transitions are allowed, the instruction stored in $IR_{i-1,l}$ $(1 \leq l \leq p)$ at time $t$ will be duplicated, and stored into both $IR_{i-1,l}$ and $IR_{i,j}$ in the next cycle. Therefore, the instruction will be executed more than once. Formally, the equation 16 should be satisfied. Similarly, the equations (equations 17–28) should be satisfied for $IR_{i,j}$. The detailed explanation is available in [16].

$$\left( \bigwedge_{l=1}^{p} cond_{IR_{i-1,l}}^{ST} \right) \wedge cond_{IR_{i,j}}^{NF} = 0 \tag{16}$$

$$cond_{IR_{i,j}}^{NF} \wedge \left( \bigwedge_{k=1}^{q} cond_{IR_{i+1,k}}^{ST} \right) = 0 \tag{17}$$

$$cond_{IR_{i,j}}^{BI} \wedge \left( \bigwedge_{k=1}^{q} cond_{IR_{i+1,k}}^{ST} \right) = 0 \tag{18}$$

$$cond_{IR_{i-1,l}}^{NF} \wedge cond_{IR_{i,j}}^{BI} = 0 \tag{19}$$

$$cond_{IR_{i-1,l}}^{BI} \wedge cond_{IR_{i,j}}^{BI} = 0 \tag{20}$$

$$cond_{IR_{i-1,l}}^{ST} \wedge cond_{IR_{i,j}}^{SQ} = 0 \tag{21}$$

$$cond_{IR_{i-1,l}}^{XN} \wedge cond_{IR_{i,j}}^{SQ} = 0 \tag{22}$$

$$cond_{IR_{i,j}}^{SQ} \wedge cond_{IR_{i+1,k}}^{NF} = 0 \tag{23}$$

$$cond_{IR_{i,j}}^{SQ} \wedge cond_{IR_{i+1,k}}^{NI} = 0 \tag{24}$$

$$cond_{IR_{i-1,l}}^{NF} \wedge cond_{IR_{i,j}}^{XN} = 0 \tag{25}$$

$$cond_{IR_{i-1,l}}^{ST} \wedge cond_{IR_{i,j}}^{XN} = 0 \tag{26}$$

$$cond_{IR_{i-1,l}}^{SQ} \wedge cond_{IR_{i,j}}^{XN} = 0 \tag{27}$$

$$cond_{IR_{i-1,l}}^{BI} \wedge cond_{IR_{i,j}}^{XN} = 0 \tag{28}$$

The above equations are not sufficient to ensure in-order execution in fragmented pipelines. An instruction $I_a$ should not reach join node earlier than an instruction $I_b$ when $I_a$ is issued by the corresponding fork node later than $I_b$. Formally the following equation should hold:

$$\forall (F, J), I_a \preceq_J I_b \Rightarrow \Gamma_F(I_a) < \Gamma_F(I_b) \tag{29}$$

where $(F, J)$ is fork-join pair, $I_a \preceq_J I_b$ implies $I_a$ reached join node $J$ before $I_b$, $\Gamma_F(I_a)$ and $\Gamma_F(I_b)$ returns the timestamps when instructions $I_a$ and $I_b$ (respectively) are issued by the fork node $F$.

The previous property ensures that instruction does not execute out-of-order. However, with the current modeling two instructions with different timestamp can reach the join node. If join node does not have capacity for more than one instruction this may cause instruction loss. We need the following property to ensure that only one immediate parent of the join node is in normal flow at time $t$:

$$\forall x, y(x, y \in \{1, 2, \ldots, p\} \wedge x \neq y),\ cond_{IR_{i-1,x}}^{NF} \wedge cond_{IR_{i-1,y}}^{NF} = 0 \tag{30}$$

Similarly, the state transition of $PC$ must depend on the state transition of $IR_{1,j}$ $(1 \leq j \leq n_1)$. The illegal combination of state transitions between $PC$ and $IR_1$,j are described below.

$$cond_{PC}^{ST} \wedge cond_{IR_{1,j}}^{NF} = 0 \tag{31}$$

$$cond_{PC}^{SE} \wedge \left( \bigwedge_{j=1}^{n_1} cond_{IR_{1,j}}^{ST} \right) = 0 \tag{32}$$

$$cond_{PC}^{BT} \wedge \left( \bigwedge_{j=1}^{n_1} cond_{IR_{1,j}}^{ST} \right) = 0 \tag{33}$$

$$cond_{PC}^{SE} \wedge cond_{IR_{1,j}}^{BI} = 0 \tag{34}$$

$$cond_{PC}^{BT} \wedge cond_{IR_{1,j}}^{BI} = 0 \tag{35}$$

$$cond_{PC}^{SE} \wedge cond_{IR_{1,j}}^{XN} = 0 \tag{36}$$

$$cond_{PC}^{ST} \wedge cond_{IR_{1,j}}^{SQ} = 0 \tag{37}$$

$$cond_{PC}^{ST} \wedge cond_{IR_{1,j}}^{XN} = 0 \tag{38}$$

We have described all possible illegal combination of state transition functions (equations 16–38). However, equations 19, 20, 23, and 24 are not necessary to prove in-order execution.

### 5.2. *Automatic Validation Framework*

Algorithm 1 describes the specification validation technique. It accepts the processor specification, described in EXPRESSION ADL [1], as input. The FSM model and the properties are generated from the ADL specification. In case of a failure, it generates counter-examples so that the designer can modify the ADL specification of the processor architecture.

**Algorithm 1**: *Validation of Pipeline Specification*
**Input**: ADL specification of the processor architecture.
**Outputs**: *Success*, if the processor model satisfies the properties.
*Failure* otherwise, and produces the counter-examples.
**Begin**
Generate FSM model from the ADL specification using equations 1–11
Generate properties using equations 12–38
Apply the properties on the FSM model to verify determinism and in-order execution.
**Return**: *Success* if all the properties are verified;
*Failure* otherwise, and produce the counter-example(s).
**End**

We have verified the properties using two different approaches. First, we have used SMV [23] based property checking framework as shown in Figure 3(a). The SMV based approach fits nicely in our validation framework. However, the SMV is limited by the size of design it can handle. We have also developed an equation solver based framework as shown in Figure 3(b) that can handle complex designs. In this section, we briefly describe these two approaches used in our framework. The detailed description is available in [16].

#### 5.2.1. *Validation using Model Checker*

The FSM model (SMV description) of the processor is generated from the ADL specification. The properties are also described using SMV description. The properties are applied on the FSM model using the SMV model checker as shown in Figure 3(a). In case of failure, SMV generates counter examples that can be used to

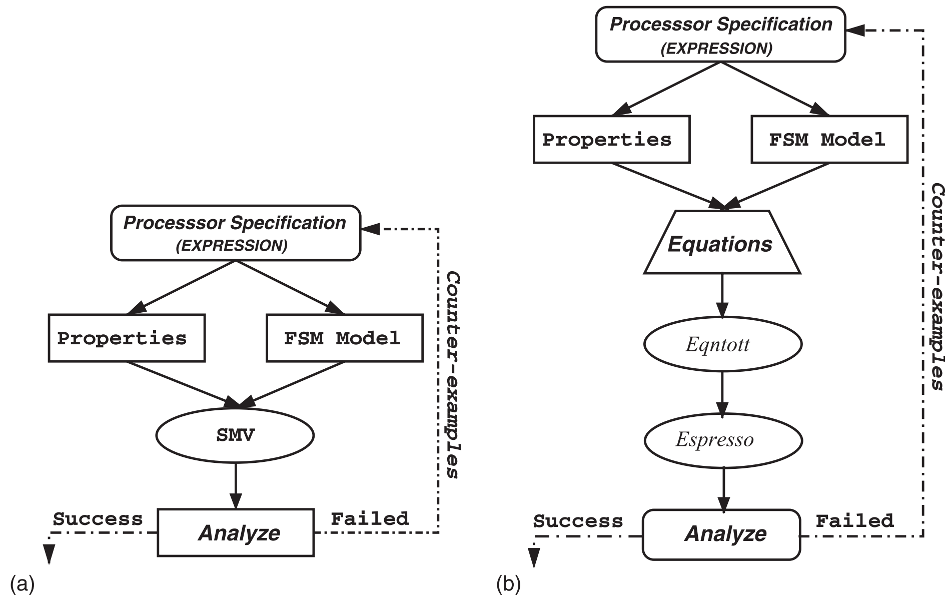


*Figure 3.* Automatic validation frameworks. (a) Framework using SMV. (b) Framework using Equation Solver.

modify the ADL specification. Each counter example describes the failed equation(s) and the instruction registers that are involved.

We have verified the in-order execution style of the processor specification in two ways. First, the framework generates properties using equations 16–38 to verify in-order execution. This is similar to how other properties (e.g., determinism) are verified. Second, an auxiliary automata is used instead of using equations to verify in-order execution. In auxiliary automata based approach, we use the same FSM model of the processor (SMV description) generated from the ADL. We developed a SMV module that generates two instructions randomly with random delay between them. These two instructions are recorded and fed to the FSM model. The processor (FSM) model accepts these instructions and performs regular computations. At the completion (e.g., writeback unit) the auxiliary automata analyzes these two instructions to see whether they completed in the same sequence as generated. Note that, this auxiliary automata does not need any manual modification for different architectures. In case of failure, SMV generates counter-examples containing instruction sequence (instruction pair with NOPs in between them) that violates in-order execution for the processor model.

### *5.2.2. Validation using Equation Solver*

In the second approach, the framework generates the FSM model and flow equations for NF, ST, XN, SQ, and BI for each instruction register and SE, ST, and BT for PC

using ADL description and equations 1–11. It generates the equations necessary for verifying properties using ADL description and equations 12–38 as shown in Figure 3(b).

The *Eqntott* [27] tool converts these equations in two-level representation of a two-valued Boolean function. This two-level representation is fed to *Espresso* [26] tool that produces minimal equivalent representation. Finally, the minimized representation is analyzed to determine whether the property is successfully verified or not. In case of failure, it generates traces explaining the cause of failure. The trace contains the equation(s) that failed, and the identification of the instruction registers involved. The designer knows the property that is violated and the reason of the violation. This information is used to modify the ADL specification. The detailed description is available in [16].

## 6. A Case Study

In a case study we successfully applied the proposed methodology to the single-issue DLX [7] processor. We have chosen DLX processor since it has been well studied in academia and contains many interesting features such as, fragmented pipelines and multicycle units that are representative of many commercial pipelined processor architectures such as TI C6x, PowerPC, and MIPS R10K.

We used the EXPRESSION ADL [1] to capture the structure and behavior of the DLX processor. We captured the conditions for stalling, normal flow, exception, branch taken, squashing, and bubble insertion in the ADL. Using the ADL description, we automatically generated the equations for flow conditions for all the units [16]. The necessary equations for verifying the properties such as, determinism and in-order execution are generated automatically from the given ADL specification. The detailed description of the case study is available in [16].

We have used *Espresso* [26] to minimize the equations. These minimized equations are analyzed to verify whether the properties are violated or not. Our framework determined that the equation 29 is violated and generated a simple instruction sequence which violates in-order execution: floating-point addition followed by integer addition. The decode unit issued floating point addition $I_{fadd}$ operation in cycle $m$ to floating-point adder pipeline and an integer addition operation $I_{iadd}$ to integer ALU at cycle $m + 1$. The instruction $I_{iadd}$ reached join node (MEM unit) prior to $I_{fadd}$.

We modified the ADL description to change the stall condition depending on current instruction in decode unit and the instructions active in the integer ALU, MUL, FADD, and DIV pipelines. The current instruction will not be issued (decode stalls) if it leads to out-of-order execution. Our framework generated equations for the modified processor model. Equation 30 is violated for this modeling for the join node (MEM unit). The instruction sequence generated by our framework for this failure consists of a multiplication operation (issued by decode unit in cycle $m$) followed by a floating-point add operation (issued by decode unit in cycle $(m + 3)$). As a result both the operations reached memory stage at cycle $(m + 7)$.

*Table 1.* Validation of In-order execution by Two Frameworks

| | DLX Processor Configurations | | |
|---|---|---|---|
| | **8 opcodes (s)** | **16 opcodes (s)** | **32 opcodes (s)** |
| SMV-based framework | 302.4 s | 400.4 s | 740.9 s |
| Espresso-based Framework | 5.4 s | 6.7 s | 9.4 s |

Finally, the stall condition of the decode unit is modified to avoid completion of two instructions at the same time. The in-order execution was successful for this modeling. In such a simple situation this kind of specification mistakes might appear as trivial, but when the architecture gets complicated and exploration iterations and varieties increases, the potential for introducing bugs also increases.

We have verified the properties using two different methods: using *SMV* model checker and *Espresso* equation solver, as described in Section 5.2. We have used 296 MHz Sun UltraSparc-II with 1024 M RAM to run the experiments. Table 1 shows the performance of the two methods for verifying in-order execution property. We have used the VLIW DLX architecture as the base configuration and modified the number of opcodes. The first column presents our two methods of specification validation. The second, third, and fourth columns present the execution time (in seconds) of the two methods for verifying in-order execution property for different architecture configurations.

We have performed experiments by modifying the pipeline structure: adding pipeline paths, adding pipeline stages etc. However, our SMV based framework could not handle configurations when pipeline path is added to the VLIW DLX architecture. Our equation solver based framework can handle complex configurations. However, for verifying the determinism (uses local computations) the SMV based framework performed better. The SMV based framework took 0.8 s to verify determinism property, whereas the equation solver based framework took 4 s for the same DLX configuration.

## 7. Conclusions

This paper proposed a framework for automatic modeling and validation of pipelined processor specifications driven by an ADL. It is necessary to validate the ADL specification of the architecture to ensure the correctness of both the architecture specified, as well as the generated software toolkit.

We have developed validation techniques to ensure that the static behavior of the pipeline is well-formed by analyzing the structural aspects of the specification using a graph-based model [19]. In this paper, we verify the dynamic behavior by analyzing the instruction flow in the pipeline using a FSM based model to validate several important architectural properties such as determinism and in-order execution in the presence of hazards and multiple exceptions. These properties are by no means

complete to prove the correctness of the specification. The designer can add new architecture specific properties and easily integrate it in our framework. Our validation framework uses two approaches: SMV based property checking and Espresso based equation minimization. The framework determines if all the necessary properties are satisfied or not. In case of a failure, it generates traces so that designer can modify the ADL specification of the architecture. We applied this methodology to the DLX processor to demonstrate the usefulness of our approach.

Currently, we can model and verify single-issue microprocessors with in-order execution, fragmented pipelines and multicycle functional units [17] in the presence of hazards and multiple exceptions [18]. Our future work will extend modeling and validation technique towards superscalar processors.

## Acknowledgements

This work was partially supported by NSF grants CCR-0203813 and CCR-0205712. We thank Prof. Sandeep Shukla and members of the ACES laboratory for their helpful comments and suggestions.

## References

1. Halambi, A. et al. EXPRESSION: A Language for Architecture Exploration Through Compiler/Simulator Retargetability. *Design Automation and Test in Europe (DATE)*, pp. 485–490, 1999.
2. Jacobi, C. Formal Verification of Complex Out-of-Order Pipelines by Combining Model-Checking and Theorem-Proving. *Computer Aided Verification (CAV)*, pp. 309–323, 2002.
3. Cyrluk, D. Microprocessor Verification in PVS: A Methodology and Simple Example. Technical Report, SRI-CSL-93-12, 1993.
4. Hadjiyiannis, G. et al. ISDL: An Instruction Set Description Language for Retargetability. *Design Automation Conference (DAC)*, pp. 299–302, 1997.
5. Burch, J. et al. Automatic Verification of Pipelined Microprocessor Control. *Computer Aided Verification (CAV)*, pp. 68–80, 1994.
6. Huggins, J. and D. Campenhout. Specification and Verification of Pipelining in the ARM2 RISC Microprocessor. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 3, no. 4, pp. 563–580, October 1998.
7. Hennessy, J. and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc, 1990.
8. Levitt, J. and K. Olukotun. Verifying Correct Pipeline Implementation for Microprocessors. *International Conference on Computer-Aided Design (ICCAD)*, p. 162–169, 1997.
9. Sawada, J. and W. D. Hunt. Processor Verification with Precise Exceptions and Speculative Execution. *Computer Aided Verification (CAV)*, pp. 364–375, 1998.
10. Skakkebaek, J. and R. Jones, and D. Dill. Formal Verification of Out-of-Order Execution Using Incremental Flushing. *Computer Aided Verification (CAV)*, pp. 98–109, 1998.
11. Aagaard, M., B. Cook, N. Day, and R. Jones. A Framework for Microprocessor Correctness Statements. *Correct Hardware Design and Verification Methods (CHARME)*, pp. 433–448, 2001.
12. Freericks, M. The nML Machine Description Formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Dept., 1993.

13. Srivas, M. and M. Bickford. Formal Verification of a Pipelined Microprocessor. *IEEE Software*, vol. 7, no. 5, pp. 52–64, 1990.
14. Velev, M. and R. Bryant. Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction. *Design Automation Conference (DAC)*, pp. 112–117, 2000.
15. Ho, P. et al. Formal Verification of Pipeline Control Using Controlled Token Nets and Abstract Interpretation. *International Conference on Computer-Aided Design (ICCAD)*, pp. 529–536, 1998.
16. Mishra, P. et al. Architecture Description Language driven Validation of Dynamic Behavior in Pipelined Processor Specifications. CECS TR 03-25, University of California, Irvine, 2003.
17. Mishra, P. et al. Automatic Verification of In-Order Execution in Microprocessors with Fragmented Pipelines and Multicycle Functional Units. *Design Automation and Test in Europe (DATE)*, pp. 36–43, 2002.
18. Mishra, P. and N. Dutt. Modeling and Verification of Pipelined Embedded Processors in the Presence of Hazards and Exceptions. *IFIP WCC Distributed and Parallel Embedded Systems (DIPES)*, pp. 81–90, 2002.
19. Mishra, P. et al. Automatic Modeling and Validation of Pipeline Specifications Driven by an Architecture Description Language. *Asia South Pacific Design Automation Conference (ASPDAC)/ International Conference on VLSI Design*, pp. 458–463, 2002.
20. Jhala, R. and K. L. McMillan. Microarchitecture Verification by Compositional Model Checking. *Computer Aided Verification (CAV)*, pp. 396–410, 2001.
21. Leupers, R. and P. Marwedel. Retargetable Code Generation based on Structural Processor Descriptions. *Design Automation for Embedded Systems*, vol. 3, no. 1, 1998.
22. Hosabettu, R. M. Systematic Verification of Pipelined Microprocessors. Ph.D. Thesis. Department of Computer Science, University of Utah, 2000.
23. *Symbolic Model Verifier*, http://www.cs.cmu.edu/~modelcheck.
24. Trimaran Release: http://www.trimaran.org. The MDES User Manual, 1997.
25. Zivojnovic, V. et al. LISA—Machine Description Language and Generic Machine Model for HW/SW Co-Design. *IEEE Workshop on VLSI Signal Processing*, 1996.
26. *Espresso*, http://www-cad.eecs.berkeley.edu/Software/software.html.
27. *Eqntott*, http://www-ee.engr.ccny.cuny.edu/notes/ee210/eqntott_man.html.