UC Irvine
ICS Technical Reports

Title

Architecture description language driven verification of in-order execution in pipelined processors

Permalink
https://escholarship.org/uc/item/95x920pf

Authors
Mishra, Prabhat
Tomiyama, Hiroyuki
Dutt, Nikil
et al.

Publication Date
2001

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

ICS

TECHNICAL REPORT

Architecture Description Language driven Verification of In-Order Execution in Pipelined Processors

Prabhat Mishra, Hiroyuki Tomiyama, Nikil Dutt, and Alex Nicolau
{pmishra, tomiyama, dutt, nicolau}@ics.uci.edu
http://www.cecs.uci.edu/~aces

UCI-ICS Technical Report #01-20
Dept. of Information and Computer Science
University of California, Irvine, CA 92697

May, 2001

Information and Computer Science
University of California, Irvine

Architecture Description Language driven Verification of In-Order Execution in Pipelined Processors

Prabhat Mishra pmishra@ics.uci.edu
Hiroyuki Tomiyama tomiyama@ics.uci.edu
Nikil Dutt dutt@ics.uci.edu
Alex Nicolau nicolau@ics.uci.edu

Architectures and Compilers for Embedded Systems (ACES) Laboratory
Center for Embedded Computer Systems
University of California, Irvine, CA, USA
http://www.cecs.uci.edu/~aces

Technical Report #01-20
Dept. of Information and Computer Science
University of California, Irvine, CA 92697, USA

May 1, 2001

Abstract

As embedded systems continue to face increasingly higher performance requirements, deeply pipelined processor architectures are being employed to meet desired system performance. System architects critically need modeling techniques that allow exploration, evaluation, customization and validation of different processor pipeline configurations, tuned for a specific application domain. We propose a novel FSM-based modeling of pipelined processors and define a set of properties that can be used to verify the correctness of in-order execution in the pipeline. Our approach leverages the system architect's knowledge about the behavior of the pipelined processor (through our ADL constructs) and thus allows a powerful top-down approach to pipeline verification.

RECEIVED
APR 15 2002
UCI LIBRARY

Contents

1 Introduction **3**

2 Related Work **3**

3 Our Approach **4**

4 Modeling of Processor Pipelines **4**
- 4.1 Processor Pipeline Description in ADL 4
- 4.2 FSM Model of Processor Pipelines 7

5 Verification of In-Order Execution **10**
- 5.1 Determinism 10
- 5.2 In-Order Execution 11
- 5.3 Finiteness 13

6 Property Verification Framework **13**
- 6.1 EXPRESSION ADL 13
- 6.2 Graph Model 15
- 6.3 FSM Model 15
- 6.4 Verify Properties 16

7 A Case Study **17**

8 Summary **21**

9 Acknowledgments **21**

List of Figures

1 The Flow in our approach 5
2 A fragment of the processor pipeline 6
3 FSM model of the fragment in Figure 2 7
4 Property Verification Framework 14
5 The DLX Processor 18

1 Introduction

Embedded systems present a tremendous opportunity to customize the designs by exploiting the application behavior using customizable processor cores and a variety of memory configurations along with different compiler techniques to meet the diverse requirements, viz., better performance, low power, smaller area, higher code density etc. However, shrinking time-to-market, coupled with increasingly short product lifetimes create a critical need to rapidly explore and evaluate candidate SOC architectures. To enable rapid design space exploration there is a need for rapid software toolkit generation. Recent work on language-driven Design Space Exploration (DSE) ([1], [3], [4], [5], [6], [8], [15], [17], [19]), uses Architectural Description Languages (ADL) to capture the processor architecture, generate automatically a software toolkit (including compiler, simulator, assembler) for that processor, and provide feedback to the designer on the quality of the architecture. It is important to verify the ADL description of the architecture to ensure the correctness of the software toolkit. The benefits of verification are two-fold. One, specification of architectures in ADLs is a tedious and error-prone process and verification techniques can be used to check for correctness of specification. Second, changes made to the processor during DSE may result in incorrect execution of the system and verification techniques can be used to ensure correctness of the architecture.

Many existing approaches ([12], [10], [20]) employ a bottom-up approach to pipeline verification/validation, where the functionality of an existing pipelined processor is, in essence, reverse-engineered from its RT-level implementation. Our approach leverages the system architects knowledge about the behavior of the pipelined processor (through our ADL constructs) and thus allows a powerful top-down approach to pipeline validation and verification, using behavioral knowledge of the pipelined architecture.

The rest of the paper is organized as follows. Section 2 presents related work addressing verification of pipelined processors. Section 3 outlines our approach and the overall flow of our environment. Section 4 presents our FSM based modeling of pipelined processors. Section 5 proposes our verification technique followed by a case study in Section 7. Section 8 concludes the paper.

2 Related Work

So far formal or semi-formal verification of pipelined processors has been studied in a number of literature. For example, Burch and Dill presented a technique for formally verifying pipelined processor control circuitry [2]. 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 [16, 21]. Huggins and Campenhout verified the ARM2 pipelined processor using Abstract State Machine [9]. In [14], Levitt and Olukotun presented a verification technique, called unpipelining, which repeatedly merges last two pipe stages into one single stage, resulting in a sequential version of the processor. Hauke and Hayes proposed a technique, called reverse engineering, which extracts the ISA model of a pipelined processor

from its implementation model [10]. Then, the extracted ISA is compared with a user-specified ISA. All the above techniques tried 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. On the other hand, in our verification approach, we are trying to define a set of properties which have to be satisfied for the correct pipeline behavior, and verify the correctness of pipelined processors by testing whether the properties are met using a Finite State Machine (FSM)-based modeling.

Iwashita et al. [13] and Ur and Yadin [20] presented pipelined processor modelings based on FSM. They used their FSM to automatically generate test programs for simulation-based validation of the processors. On the other hand, this paper addresses formal verification of pipelined processors without simulation.

Tomiyama et al. [18] presented FSM based modeling of pipelined processors with in-order execution and closest to our approach. Their model can handle only simple processors with straight pipeline. On the other hand, our model can handle processors with fragmented pipelines and multicycle units. They defined three properties that need to be met for correct in-order execution. However, the paper did not describe how to apply these properties for verifying the correctness. In our verification approach, we present an automatic property checking framework driven by an ADL.

3 Our Approach

Figure 1 shows the flow in our approach. In our IP library based exploration and verification scenario, the designer starts by specifying the processor and memory subsystem description in ADL. The FSM model of the pipelined processor description is automatically generated from the ADL description. We have defined properties to ensure that the ADL description of the architecture is well-formed. Our automatic property checking framework determines if the property is satisfied or not. In case of failure, it generates the traces so that the designer can modify the ADL specification of the architecture. If the verification is successful, the software toolkit (including compiler and simulator) can be generated for design space exploration.

4 Modeling of Processor Pipelines

In this section we describe how we model the pipeline in FSM from the ADL description of the processor. We first explain the information captured in the ADL necessary for the 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

An ADL that contains a description of both the behavior and the structure of the processor can be used in our verification and exploration framework. The advantage of using mixed-level ADLs is that it becomes possible to verify the structure against the behavior (e.g., verification of the

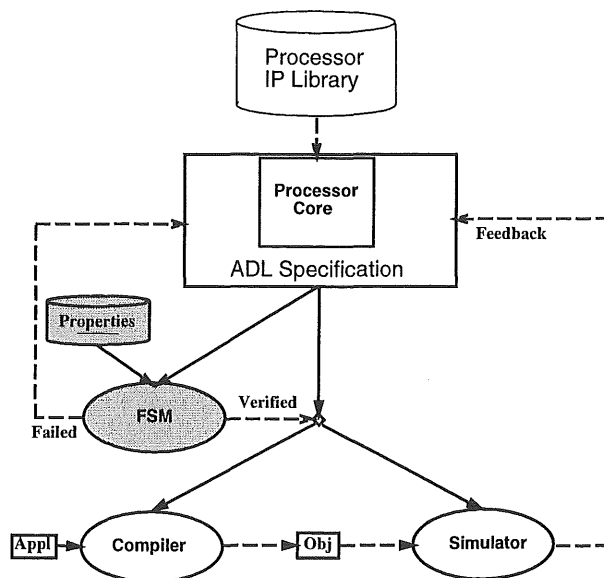

Figure 1. The Flow in our approach

pipeline structure against the expected behavior). We first describe the structure and behavior of the processor captured in ADL, then we describe how to capture the conditions for stalling, normal flow and nop insertion for each functional unit.

Structure and Behavior of the Processor

The structure contains functional and storage units connected by pipeline and data transfer paths. A path from a functional unit to storage or from a storage to a functional unit is called a data transfer path. A path from the first unit (e.g, PC Unit or Fetch Unit in most of the processors) to the last unit (e.g., WriteBack Unit or Completion Unit in most of the processors) consisting of functional units and pipeline edges is called a pipeline path. A pipeline edge specifies the ordering of functional units comprising the pipeline stages. Intuitively, a pipeline path denotes an execution flow in the pipeline taken by an operation. Informally, a pipeline edge transfers instruction and data from parent unit to child unit using pipeline latch (referred in this paper as pipeline register or instruction register). Each unit may have several attributes e.g., *capacity*, *timing* etc. The capacity denotes the maximum number of operations which the unit can handle in a cycle (e.g., certain decode unit in VLIW processor can issue m operations per cycle to the m parallel execution units), while the timing denotes the number of cycles taken by the unit (e.g., certain multicycle unit takes n cycles whereas single-cycle unit takes 1 cycle).

The behavior of a processor is a set of operations that can be executed on it. Each operation in turn consists of a set of fields (e.g., opcode, arguments etc.) that specify, at an abstract level, the execution semantics of the operation.

The mapping between structure and behavior captures, for each functional or storage unit, the set of operations supported by the unit.

Modeling Conditions in ADL

Along with the structure, behavior and their mapping information already available in the ADL description, designers need to capture in ADL the conditions for stalling, normal flow and nop insertion for each functional unit.

A unit can be stalled due to signals external to the processor or due to contributions arising inside the processor. For example, the external signal that can stall the fetch unit is *ICache_Miss*; internal condition for stalling of fetch unit can be due to decode stall. For units, with multiple children the stalling conditions due to internal contribution may differ. For example, the unit $UNIT_{i-1,j}$ in Figure 2 with q children can be stalled when anyone of its children are stalled, or when some of its children are stalled (designers identifies the specific ones), or when all of its children are stalled; or when none of its children are stalled. During description, designer chooses from the set (ANY, SOME, ALL, NONE) for internal contribution along with any external signals to specify stall condition for each unit.

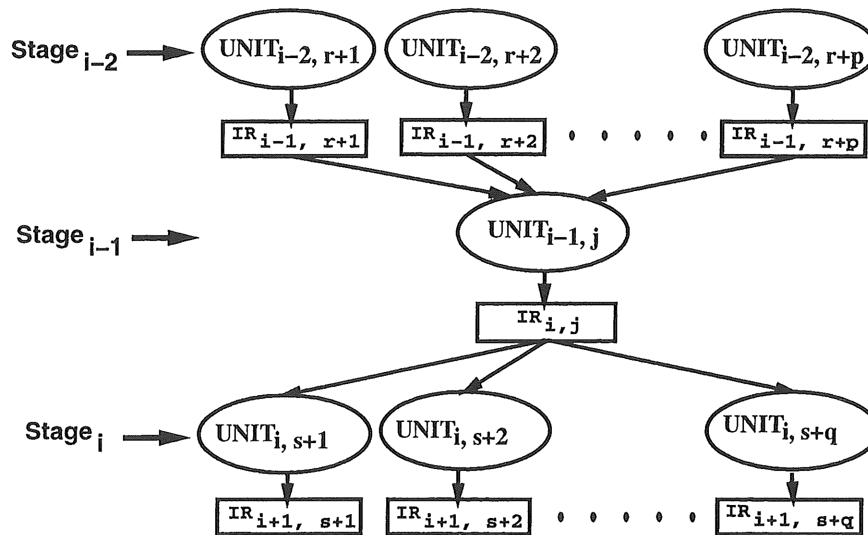

Figure 2. A fragment of the processor pipeline

A unit can be in normal flow if it can receive instruction from its parent unit and can send to its child latch. For units with multiple parents and multiple children the normal flow condition may differ. For example, the unit $UNIT_{i-1,j}$ in Figure 2 with p parent units and q children units can flow normally depending on several combination of states among its parent units and child units. For example, one of its parents is not stalled and one of its children is not stalled, or one of its parents is not stalled and all of its children are not stalled (e.g., decode stage in a VLIW processor with no reservation station) etc. During specification the designer chooses from the set (ANY, SOME, ALL, NONE) the contributions from the parents and children to specify the normal flow condition for each unit.

Typically, a unit performs nop insertion when it does not receive any instruction from the parent (or busy computing in case of multicycle unit) and its child is not stalled. For units with multiple parents and multiple children the nop insertion condition may differ. For example, the unit

$UNIT_{i-1,j}$ in Figure 2 with p parent units and q children units can perform nop insertion depending on several combination of states among its parent units and child units. For example, all of its parents are stalled and one of its children is not stalled etc. During specification the designer chooses from the set (ANY, SOME, ALL, NONE) the contributions from parents and children to specify the nop insertion condition for each unit.

4.2 FSM Model of Processor Pipelines

This section presents an FSM-based modeling of controllers in pipelined processors. This paper especially focuses on the next state function of the FSM.

We assume a pipelined processor with in-order execution as the target for modeling and verification. The pipeline consists of N stages. Each stage can have more than one pipeline register (in case of fragmented pipeline). Each single-cycle pipeline register takes one cycle if there are no pipeline hazards. Multi-cycle pipeline register takes n cycles during normal execution (no hazard). In this paper we call these pipeline registers instruction registers since they contain instructions being executed in the pipeline. 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 $Stage_i$ $(1 \leq i \leq N-1)$. Let $IR_{i,j}$ denote a instruction register between $Stage_{i-1}$ and $Stage_i$ $(1 \leq i \leq N-1,\ 1 \leq j \leq N_i)$. The first stage, i.e., $Stage_0$, fetches from instruction memory an instruction pointed by program counter PC, and stores the instruction into the first instruction register $IR_{1,j}$ $(1 \leq j \leq N_1)$. During execution the instruction stored in $IR_{i,j}$ is executed at $Stage_i$ and then stored into the next instruction register $IR_{i+1,s+k}$ $(1 \leq k \leq N_{i+1})$

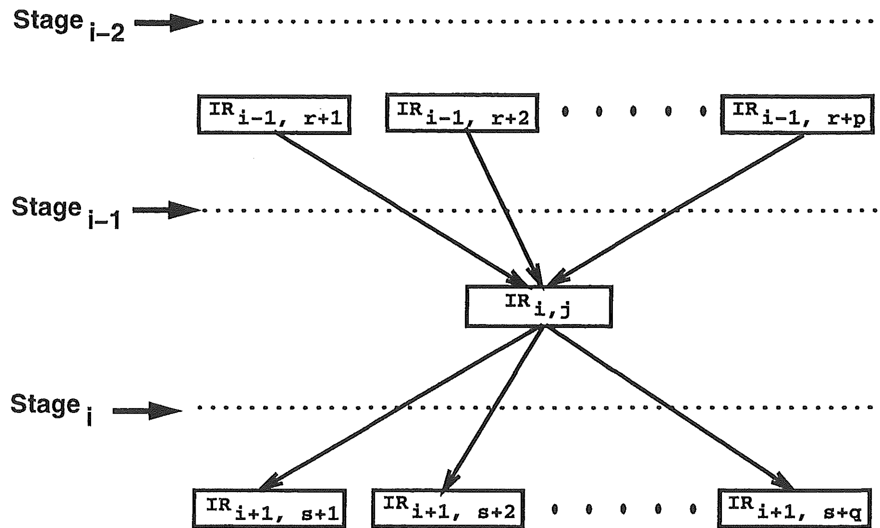

Figure 3. FSM model of the fragment in Figure 2

In this paper, we define a state of the N-stage pipeline as values of PC and $(N-1)\sum_{i=1}^{N-1} N_i$ (= M say) instruction registers$(1 \leq i \leq N-1)$. 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

$$S(t) = < PC(t), IR_{1,1}(t), \cdots, IR_{N-1,N_{N-1}}(t) > \tag{1}$$

We first explain the conditions for stalling(ST), normal flow(NF), nop insertion(NI), and branch taken (BT) in the FSM model, then we describe the state transition functions possible in the FSM model using these conditions.

Modeling conditions in FSM

Let us assume, every instruction register $IR_{i,j}$ has a stall bit $ST_{IR_{i,j}}$, which is set when the stall condition ($cond^{ST}_{IR_{i,j}}$ say) is true. As mentioned in Section 4.1, $ST_{IR_{i,j}}$ has two components viz., stall condition due to the stall of children ($ST^{child}_{IR_{i,j}}$ say) and stall condition due to external signals on $IR_{i,j}$ (($ST^{self}_{IR_{i,j}}$ say). More formally the condition for stalling at time t in the presence of a set of external signals $I(t)$ on $S(t)$ is,

$$cond^{ST}_{IR_{i,j}}(S(t), I(t)) = ST_{IR_{i,j}} = ST^{child}_{IR_{i,j}} + ST^{self}_{IR_{i,j}} \tag{2}$$

For example, if designer specified that "ALL" (see Section 4.1) the children are responsible for the stalling of $IR_{i,j}$. Then Equation (2) becomes
$cond^{ST}_{IR_{i,j}} = ST_{IR_{i,j}} = \cap^{q}_{k=0} ST_{IR_{i+1,k}} + ST^{self}_{IR_{i,j}}$

As mentioned in Section 4.1, the condition for normal flow ($cond^{NF}_{IR_{i,j}}$ say) has three components viz., contribution from parents ($NF^{parent}_{IR_{i,j}}$ say), contribution from children ($NF^{child}_{IR_{i,j}}$ say), and self contribution (not stalled). More formally,

$$cond^{NF}_{IR_{i,j}}(S(t), I(t)) = NF^{parent}_{IR_{i,j}} . NF^{child}_{IR_{i,j}} . \overline{ST^{self}_{IR_{i,j}}} \tag{3}$$

For example, if designer specified that $IR_{i,j}$ will be in normal flow if "ANY" (see Section 4.1) of the parents is not stalled and "ANY" of the children is not stalled. Then Equation (3) becomes
$cond^{NF}_{IR_{i,j}} = \cup^{p}_{l=1} \overline{ST_{IR_{i-1,r+l}}} . \cup^{q}_{k=1} \overline{ST_{IR_{i+1,r+k}}} . \overline{ST^{self}_{IR_{i,j}}}$

As mentioned in Section 4.1, the condition for nop insertion ($cond^{NI}_{IR_{i,j}}$ say) has three components viz., contribution from parents ($NI^{parent}_{IR_{i,j}}$ say), contribution from children ($NI^{child}_{IR_{i,j}}$ say), and self contribution (not stalled). More formally,

$$cond^{NI}_{IR_{i,j}}(S(t), I(t)) = NI^{parent}_{IR_{i,j}} . NI^{child}_{IR_{i,j}} . \overline{ST^{self}_{IR_{i,j}}} \tag{4}$$

For example, if designer specified that $IR_{i,j}$ will be in nop insertion if "ALL" (see Section 4.1) of the parents are stalled and "ANY" of the children is not stalled. Then Equation (4) becomes
$cond^{NI}_{IR_{i,j}} = \cap^{p}_{l=1} ST_{IR_{i-1,s+l}} . \cup^{q}_{k=1} \overline{ST_{IR_{i+1,s+k}}} . \overline{ST^{self}_{IR_{i,j}}}$

Similarly the conditions for PC viz., $cond_{PC}^{SE}$(SE: sequential execution, $cond_{PC}^{NI}$, and $cond_{PC}^{BT}$ (BT: branch taken) can be described using the information available in the ADL. Let us assume, BT_{PC} bit is set when the unit completes execution of a branch instruction. Formally,

$$cond_{PC}^{SE}(S(t),I(t)) = NF_{PC}^{child}.\overline{ST_{PC}^{self}}.\overline{BT_{PC}} \tag{5}$$

$$cond_{PC}^{ST}(S(t),I(t)) = (ST_{PC}^{child} + ST_{PC}^{self}).\overline{BT_{PC}} \tag{6}$$

$$cond_{PC}^{BT}(S(t),I(t)) = BT_{PC} \tag{7}$$

Modeling State Transition Functions

In this section, we describe the next state function of the FSM. Figure 3 shows the FSM model of the fragment of the processor pipeline shown in Figure 2. If there are no pipeline hazards, instructions flow from IR (instruction register) to IR every n cycles (n = 1 for single-cycle IR). In this case, the instruction in $IR_{i-1,r+l}$ $(1 \leq l \leq p)$ at time t proceeds to $IR_{i,j}$ after n cycles (n is the *timing* of $IR_{i-1,r+l}$ and $IR_{i,j}$ has p parent latches and q child latches as shown in Figure 3), i.e., $IR_{i,j}(t+1) = IR_{i-1,r+l}(t)$. In presence of pipeline hazards, however, the instruction in $IR_{i,j}$ may be stalled, i.e., $IR_{i,j}(t+1) = IR_{i,j}(t)$. It should be also noted that, in general, any instruction in the pipeline cannot skip pipe stages. This means that, for example, $IR_{i,j}(t+1)$ cannot be $IR_{i-2,v}(t)$ $(1 \leq v \leq N_{i-2})$. Now we can easily understand that there are some specific rules which must be followed in the next state function of the FSM.

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 $(M+1)$ $(M = (N-1)\sum_{i=1}^{N-1} N_i)$ registers. 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 $f_{IR_{i,j}}^{NS}$ $(1 \leq i \leq N-1,\ 1 \leq j \leq N_i)$ 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.

$$\begin{aligned} & PC(t+1) \\ &= f_{PC}^{NS}(S(t),I(t)) \\ &= \begin{cases} PC(t)+L & \text{if } cond_{PC}^{SE}(S(t),I(t)) = 1 \\ target & \text{if } cond_{PC}^{BT}(S(t),I(t)) = 1 \\ PC(t) & \text{if } cond_{PC}^{ST}(S(t),I(t)) = 1 \end{cases} \end{aligned} \tag{8}$$

Here, $I(t)$ represents a set of external signals at time t, L the instruction length (4 bytes in many processors), and *target* the branch target address which is computed at a certain pipe stage. $cond_{PC}$'s are logic functions of $S(t)$ and $I(t)$ as described in Section 4.2, and return either 0 or 1. If $cond_{PC}^{ST}(S(t),I(t))$ is 1, PC keeps its current value at the next cycle.

For the instruction register, $IR_{1,j}$ $(1 \leq j \leq N_1)$, we define the following three types of state transitions.

$$IR_{1,j}(t+1)$$

$$
\begin{aligned}
&= f^{NS}_{IR_{1,j}}(S(t),I(t)) \\
&= \begin{cases} IM(PC(t)) & \text{if } cond^{NF}_{IR_{1,j}}(S(t),I(t)) = 1 \\ IR_{1,j}(t) & \text{if } cond^{ST}_{IR_{1,j}}(S(t),I(t)) = 1 \\ nop & \text{if } cond^{NI}_{IR_{1,j}}(S(t),I(t)) = 1 \end{cases}
\end{aligned} \tag{9}
$$

Similarly, for the other instruction register, $IR_{i,j}$ ($2 \leq i \leq N-1$, $1 \leq j \leq N_i$), we define three types of state transitions as follows.

$$
\begin{aligned}
&IR_{i,j}(t+1) \\
&\quad = f^{NS}_{i,j}(S(t),I(t)) \\
&\quad = \begin{cases} IR_{i-1,l}(t) & \text{if } cond^{NF}_{IR_{i,j}}(S(t),I(t)) = 1 \\ IR_{i,j}(t) & \text{if } cond^{ST}_{IR_{i,j}}(S(t),I(t)) = 1 \\ nop & \text{if } cond^{NI}_{IR_{i,j}}(S(t),I(t)) = 1 \end{cases}
\end{aligned} \tag{10}
$$

In the above formulas, nop denotes a special instruction indicating that there is no instruction in the instruction register, and $IM(PC(t))$ denotes the instruction pointed by the program counter. If $cond^{NF}_{IR_{1,1}}(S(t),I(t))$ is 1, an instruction is fetched from instruction memory and stored into $IR_{1,1}$. If $cond^{ST}_{IR_{1,1}}(S(t),I(t))$ is 1, $IR_{1,1}$ remains unchanged. In this paper, $IR_{i,j}$ is said to be stalled at time t if $cond^{ST}_{IR_{i,j}}(S(t),I(t))$ 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 $cond^{NF}_{IR_{i,j}}(S(t),I(t))$ is 1. A nop instruction is inserted in $IR_{i,j}$ when $cond^{ST}_{IR_{i,j}}(S(t),I(t))$ is 1, resulting in $IR_{i,j}(t+1) = nop$.

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

5 Verification of In-Order Execution

Based on the FSM modeling presented in the previous section, we propose a method to verify the correctness of controllers of pipelined processors with in-order execution. In this paper we define that an in-order pipelined processor is correct if all instructions which are fetched from instruction memory flow from the first stage to the last stage with keeping their execution order. This section presents three properties: determinism, in-order execution, and finiteness. If a given pipelined processor meets all these properties, the processor is guaranteed to be correct for in-order execution.

5.1 Determinism

The next state functions for all state registers must be deterministic. This property is valid if all the following equations hold.

$$cond^{SE}_{PC}(S(t),I(t)) \textbf{ or } cond^{BT}_{PC}(S(t),I(t))$$

$$\textbf{or } cond_{PC}^{ST}(S(t),I(t)) = 1 \quad (11)$$

$$cond_{IR_{i,j}}^{NF}(S(t),I(t)) \textbf{ or } cond_{IR_{i,j}}^{ST}(S(t),I(t)) \textbf{ or } cond_{IR_{i,j}}^{NI}(S(t),I(t)) = 1, \quad \forall i,j(1 \leq i \leq N-1, 1 \leq j \leq N_i) \quad (12)$$

$$cond_{PC}^{x}(S(t),I(t)) \textbf{ and } cond_{PC}^{y}(S(t),I(t)) = 0, \quad \forall x,y(x,y \in \{SE,BT,ST\} \wedge x \neq y) \quad (13)$$

$$cond_{IR_{i,j}}^{x}(S(t),I(t)) \textbf{ and } cond_{IR_{i,j}}^{y}(S(t),I(t)) = 0, \quad \forall i,j(1 \leq i \leq N-1, 1 \leq j \leq N_i), \quad \forall x,y(x,y \in \{NF,ST,NI\} \wedge x \neq y) \quad (14)$$

The first two equations mean that, in the next state function for each state register, the three conditions must cover all possible combinations of processor state $S(t)$ and external signals $I(t)$. The last two guarantee that any two conditions are disjointed for each next state function. Informally, exactly one of the conditions should be true in a cycle.

5.2 In-Order Execution

In order to guarantee in-order execution, state transitions of adjacent instruction registers must depend on each other. Illegal combinations of state transitions of adjacent stages are described below using Figure 3.

- 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 is allowed, the instruction stored in $IR_{i-1,r+l}$ $(1 \leq l \leq p)$ at time t will be duplicated, and stored into both $IR_{i-1,r+l}$ and IR_i at the next cycle. Therefore, the instruction will be executed more than once. More formally, the Equation (15) should be satisfied.

$$\cap_{l=1}^{p} cond_{IR_{i-1,r+l}}^{ST} \textbf{ and } cond_{IR_{i,j}}^{NF} = 0 \quad (2 \leq i \leq N-1, 1 \leq j \leq N_i, 1 \leq l \leq p) \quad (15)$$

- Similarly, if $IR_{i,j}$ flows normally, at least one of its child latches should also flow normally. If all of its child latches are stalled, the instruction stored in $IR_{i,j}$ disappears. More formally, the Equation (16) should be satisfied.

$$cond_{IR_{i,j}}^{NF} \textbf{ and } \cap_{k=1}^{q} cond_{IR_{i+1,s+k}}^{ST} = 0 \quad (2 \leq i \leq N-1, 1 \leq j \leq N_i, 1 \leq k \leq q) \quad (16)$$

- Similarly, if $IR_{i,j}$ is in nop insertion, at least one of its child latches should not be stalled. If all of its child latches are stalled, the instruction stored in $IR_{i-1,r+l}$ $(1 \leq l \leq p)$ at time t will be overwritten by the nop instruction. More formally, the Equation (17) should be satisfied.

$$cond^{NI}_{IR_{i,j}} \textbf{ and } \cap_{k=1}^{q} cond^{ST}_{IR_{i+1,s+k}} = 0 \quad (2 \leq i \leq N-1, 1 \leq j \leq N_i, 1 \leq k \leq q) \tag{17}$$

- Similarly, an instruction register can not be in nop insertion, if previous instruction register is in normal flow. More formally, the Equation (18) should be satisfied.

$$cond^{NF}_{IR_{i-1,r+l}} \textbf{ and } cond^{NI}_{IR_{i,j}} = 0 \quad (2 \leq i \leq N-1, 1 \leq j \leq N_i, 1 \leq l \leq p) \tag{18}$$

- Finally, an instruction register can not be in nop insertion, if previous instruction register is also in nop insertion. More formally, the Equation (19) should be satisfied.

$$cond^{NI}_{IR_{i-1,r+l}} \textbf{ and } cond^{NI}_{IR_{i,j}} = 0 \quad (2 \leq i \leq N-1, 1 \leq j \leq N_i, 1 \leq l \leq p) \tag{19}$$

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. More formally the following equation should hold:

$$\forall (F,J), I_a \preceq_J I_b \Rightarrow \Gamma_F(I_a) < \Gamma_F(I_b) \tag{20}$$

where, (F, J) is fork-join pair, $I_a \preceq_J I_b$ implies I_a reached join node J before I_b, $\Gamma_F(I_a)$ returns the timestamp when instruction I_a is 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 (refer Figure 3):

$$\forall x, y (x, y \in \{1, 2, ..., p\} \wedge x \neq y) \quad cond^{NF}_{IR_{i-1,r+x}} \cdot cond^{NF}_{IR_{i-1,r+y}} = 0 \tag{21}$$

Similarly, the state transition of PC must depend on the state transition of $IR_{1,j}$ $(1 \leq j \leq N_1)$. The illegal combinations of state transitions are described below.

$$cond^{ST}_{PC} \textbf{ and } cond^{NF}_{IR_{1,j}} = 0 \tag{22}$$

$$cond^{SE}_{PC} \textbf{ and } \cap_{j=1}^{N_1} cond^{ST}_{IR_{1,j}} = 0 \tag{23}$$

$$cond^{BT}_{PC} \textbf{ and } \cap_{j=1}^{N_1} cond^{ST}_{IR_{1,j}} = 0 \tag{24}$$

$$cond^{SE}_{PC} \textbf{ and } cond^{NI}_{IR_{1,j}} = 0 \tag{25}$$

$$cond^{BT}_{PC} \textbf{ and } cond^{NI}_{IR_{1,j}} = 0 \tag{26}$$

The in-order execution property is guaranteed if all the above equations (Equation (15) - Equation (26)) hold.

5.3 Finiteness

The determinism and in-order execution properties do not guarantee that execution of instructions will be completed in a finite number of cycles. In other words, the pipeline might be stalled infinitely. Therefore, we need to guarantee that stall conditions (i.e., $cond^{ST}_{IR_{i,j}}$) are resolved in a finite number of cycles. As mentioned in Equation (2) pipeline stalls have two components. Both components must be resolved in a finite number of cycles. The following conditions are sufficient to guarantee the finiteness.

- A stage must flow within a finite number of cycles if all the later stages are idle. Since this condition may depend on external signals which come from outside of the processor core, it cannot be verified only with the FSM model. This condition is a constraint in the design of the blocks which generate such signals.
- $cond^{x}_{IR_{i,j}}$ $(x \in (NF, ST, NI))$ can be a function of external signals and/or $IR_{k,y}$ where $k \geq i$, but cannot be a function of IR_k where $k < i$.

6 Property Verification Framework

In this section we describe the automatic property verification framework as shown in Figure 4. We first describe the EXPRESSION ADL which is used to capture the programmable architectures, then we discuss how the graph model is generated automatically from the ADL description. Next, we explain how to generate finite state machine model of the processor controller automatically from the graph model. This FSM model is used for verifying the in-order execution. Finally, we present how to verify in-order execution using this FSM modeling.

6.1 EXPRESSION ADL

The EXPRESSION [8] ADL captures the structure and behavior of the processor-memory pipeline. However, for the property verification we need to capture the conditions for stalling, normal flow and nop insertion for each functional or storage unit. The syntax of the condition specification is shown below. The syntax of the structure (input/output latches, ports, capacity, timing etc.) and behavior specification are not shown here and can be found in [7].

```
(UnitType UnitName
    ......
    (CONDITIONS
        <list_of_flow_conditions>
        <SELF <list_of_external_signals>
    )
)

<list_of_flow_conditions> :=  <flow_condition>
                            | <list_of_flow_conditions>

<flow_condition> := (<flow_type> <parent_children_contributions>)

<flow_type> :=  NF  /* Normal Flow */
```

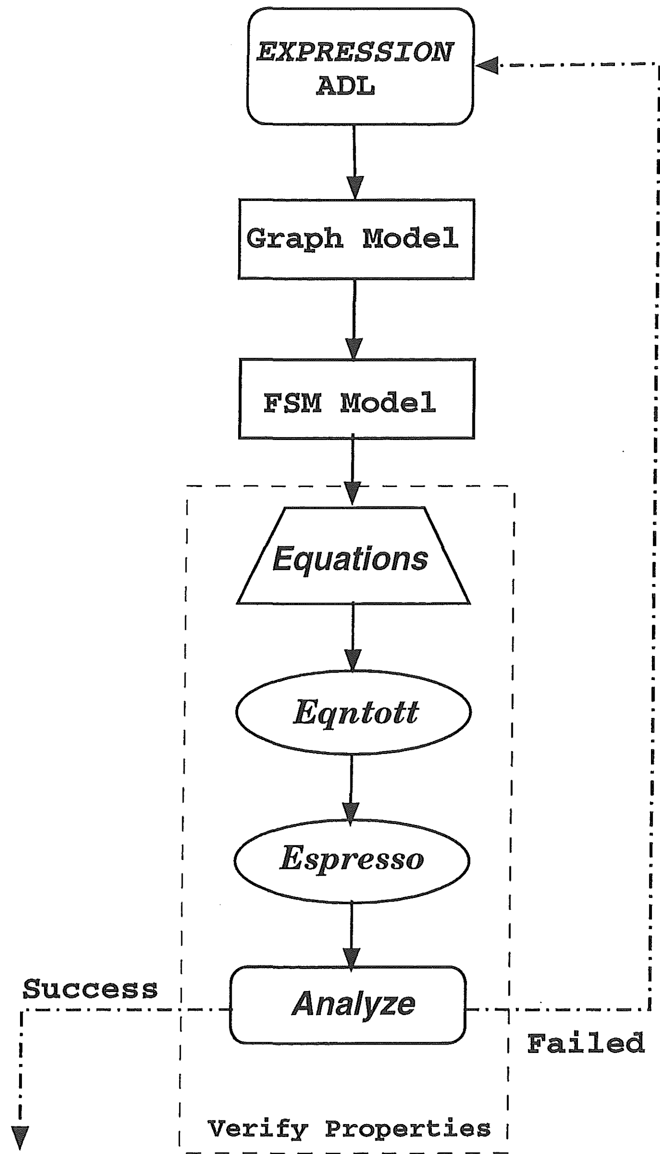

Figure 4. Property Verification Framework

```
          | NI   /* Nop Insertion */
          | BT   /* Branch Taken */
          | ST   /* Stall */

<parent_children_contributions> :=  <parent_contribution> <child_contribution>
                                  | <child_contribution>
                                  | NULL

<parent_contribution> := <contribution>
<child_contribution> := <contribution>

<contribution> :=  NONE  /* None of the parents/children can control the flow of the current node */
                 | ANY   /* Any one of the parents/children can control the flow of the current node */
                 | SOME  /* Some of the parents/children can control the flow of the current node */
                 | ALL   /* All of the parents/children can control the flow of the current node */
```

The following example shows the condition specification of the decode unit for DLX processor (Figure 5). The decode unit is in normal flow (NF) when ANY of the parents are not stalled and ANY of the children are not stalled. The decode unit is stalled when ALL of its children are stalled. The decode unit is in nop insertion when ALL of its parents are stalled and ANY of its children are not stalled. It does not get any external signal (SELF "") which can change the flow of the unit.

```
(DecodeUnit DECODE
    .........
    (CONDITIONS
        (NF ANY ANY)
        (ST ALL)
        (NI ALL ANY)
        (SELF "")
    )
)
```

6.2 Graph Model

The data structure generated after reading EXPRESSION description has connections from units to latches and latches to units or storages using ports and connections. A fragment of this structure is shown in Figure 2. We add few more connections e.g., parent pointer, list of children units for each unit etc., for making graph traversal easier for later stages.

6.3 FSM Model

Figure 3 shows the FSM model of the fragment of the graph shown in Figure 2. Each latch (instruction register) has a list of parent and child pointers. We generate the flow equations for NF, ST, BT, and NI for each latch using the conditions available in the EXPRESSION ADL. For example, the equations for the decode latch (using the description of the decode unit shown in Section 6.1 and in Figure 5) are shown below.

$$decode_NF_equation = \overline{ST_{IR_{1,1}}} \textbf{ and } (\overline{ST_{IR_{2,1}}} \textbf{ or } \overline{ST_{IR_{2,2}}} \textbf{ or } \overline{ST_{IR_{2,3}}} \textbf{ or } \overline{ST_{IR_{2,4}}}) \quad (27)$$

$$decode_ST_equation = ST_{IR_{2,1}} \textbf{ and } ST_{IR_{2,2}} \textbf{ and } ST_{IR_{2,3}} \textbf{ and } ST_{IR_{2,4}} \quad (28)$$

$$decode_NI_equation = ST_{IR_{1,1}} \textbf{ and } (\overline{ST_{IR_{2,1}}} \textbf{ or } \overline{ST_{IR_{2,2}}} \textbf{ or } \overline{ST_{IR_{2,3}}} \textbf{ or } \overline{ST_{IR_{2,4}}}) \quad (29)$$

6.4 Verify Properties

In this section we describe how the properties are verified using the FSM model described in Section 6.3. We first generate equations necessary for verifying properties as described in Section 5. We use **eqntott** tool to convert these equations in two-level representation of a two-valued boolean function. This two-level representation is fed to **espresso** tool which 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 the trace explaining the cause of failure.

Generate Equations

In this framework we verify determinism and in-order execution properties. For verifying determinism we generate the equations for each latch of the FSM model as described in Section 5.1. We generate these equations in eqntott acceptable format. For example, the determinism equations for the decode latch are shown below using the flow equations described in Equation (27) - Equation (29). The first line describes all the inputs in that particular order. The second line describes the outputs in that order. The third line describes the Equation (12). The last three lines present the Equation (14) for NF/ST, NF/NI, and ST/NI scenarios. For the ease of illustration we have used unit names instead of latch names (e.g., we used stLATCH instead of $ST_{IR_{1,1}}$).

```
INORDER =  stFETCH stIALU stM1 stA1 stFDIV;
OUTORDER = all nf_st nf_ni st_ni;
all = ( ( !stFETCH ) & ( !stIALU | !stM1 | !stA1 | !stFDIV ) ) | ( ( stIALU & stM1 & stA1 & stFDIV ) ) |
          ( ( stFETCH ) & ( !stIALU | !stM1 | !stA1 | !stFDIV ) );
nf_st = ( ( !stFETCH ) & ( !stIALU | !stM1 | !stA1 | !stFDIV ) ) & ( ( stIALU & stM1 & stA1 & stF-
DIV ) );
nf_ni = ( ( !stFETCH ) & ( !stIALU | !stM1 | !stA1 | !stFDIV ) ) &
        ( ( stFETCH ) & ( !stIALU | !stM1 | !stA1 | !stFDIV ) );
st_ni = ( ( stIALU & stM1 & stA1 & stFDIV ) ) & ( ( stFETCH ) & ( !stIALU | !stM1 | !stA1 | !stFDIV ) );
```

For verifying in-order execution we generate equations for each pair of adjacent latches as described in Section 5.2. We generate these equations in eqntott acceptable format. For example, the in-order execution equations for the decode-IALU (shown as EX in Figure 5) latch pair are shown below. The first line describes all the inputs in that particular order. The second line describes the outputs in that order. The third line describes the condition when decode is stalled. The last five lines present the interactions described in Equation (15) - Equation (19).

```
INORDER =  stFETCH stIALU stM1 stA1 stFDIV stMEM;
OUTORDER = st_nf nf_st ni_st nf_ni ni_ni stDECODE;
stDECODE = ( ( stIALU & stM1 & stA1 & stFDIV ) );
st_nf = (  stDECODE ) & ( ( !stDECODE ) & ( !stMEM ) );
nf_st = ( ( !stFETCH ) & ( !stIALU | !stM1 | !stA1 | !stFDIV ) ) & (  stIALU & stM1 & stA1 & stFDIV );
ni_st = ( ( stFETCH ) & ( !stIALU | !stM1 | !stA1 | !stFDIV ) ) & (  stIALU & stM1 & stA1 & stFDIV );
nf_ni = ( ( !stFETCH ) & ( !stIALU | !stM1 | !stA1 | !stFDIV ) ) & ( ( stDECODE ) & ( !stMEM ) );
ni_ni = ( ( stFETCH ) & ( !stIALU | !stM1 | !stA1 | !stFDIV ) ) & ( ( stDECODE ) & ( !stMEM ) );
```

Generate Truth Table

We use eqntott program to convert original equations in espresso acceptable format. Eqntott generates a truth table suitable for PLA programming from a set of Boolean equations which define the PLA outputs in terms of its inputs. We use "-s" option, since we use some of the output variables as input. This program can be found in
http://buffy.eecs.berkeley.edu/IRO/Software/Catalog/Description/platools.html

Minimize Truth Table

We use espresso program to minimize the equations that we generate for verifying the properties. Espresso takes as input a two-level representation of a two-valued (or multiple-valued) Boolean function, and produces a minimal equivalent representation. This program can be found in *http://www-cad.eecs.berkeley.edu/Software/software.html*

Analyze

In our framework, each equation is minimized to '1' or '0'. When it is is not minimized to the boolean value, analyzer declares that as a failure and produces the trace. It compares generated value ('1' or '0') with the expected one (as described in Equation (11) - Equation (26)) and determines whether the property holds true or not and generates necessary trace in case of failure.

7 A Case Study

In a case study we successfully applied the proposed methodology to the DLX [11] processor. We have chosen DLX processor since it has been well studied in academia and has few interesting features viz., fragmented pipelines, multicycle units etc. Figure 5 shows the DLX processor pipeline.

We used EXPRESSION ADL to capture the structure and behavior of the DLX processor. The necessary equations for verifying the properties viz., determinism, in-order execution etc., are generated automatically from the given ADL description. We have used espresso to minimize the equations. These minimized equations are analyzed to verify whether the properties are violated or not. In case of violation it displays the cause of failure. The framework is fully automated. The complete verification, starting from ADL description to property verification, takes 41 seconds on a 333 MHz Sun Ultra-5 with 128M RAM.

We captured the conditions for stalling, normal flow, branch taken and nop insertion in the ADL. For example, we captured *CacheMiss* as the external signal for PC unit. For all the units we assumed "ALL" contribution from the children for stall condition. While capturing normal flow condition for each unit we selected "ANY" for parent units and "ANY" for child units. Similarly, for each unit we capture "ALL" as contribution from parent units and "ANY" as contribution for child units. Using this information, we generated automatically the conditions for all the units as shown below. $IR_{2.4}$ represents latch for the multicycle unit. So we assumed a signal *busy* internal to $IR_{2.4}$ which remained set for n cycles. The *busy* can be treated as $ST_{IR_{2.4}}^{self}$ as shown in Equation (2).

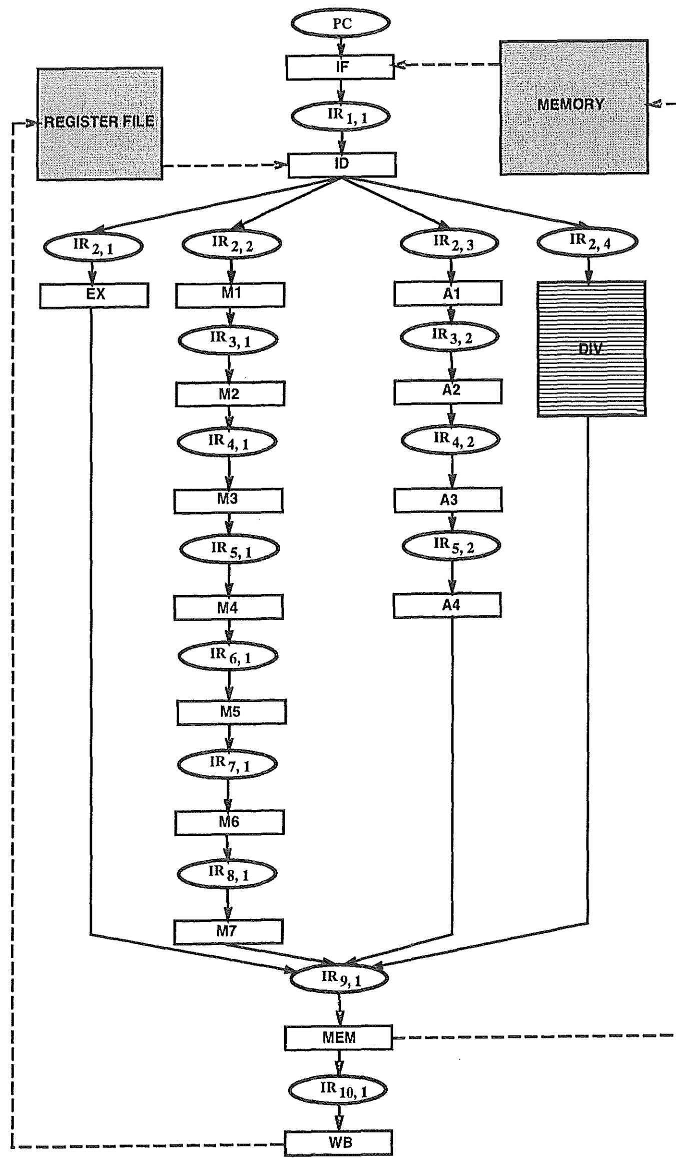

Figure 5. The DLX Processor

$$
\begin{aligned}
cond_{PC}^{NF} &= \overline{ICacheMiss}.\overline{ST_{IR_{1,1}}}.\overline{IALU.opcode == BR} \\
cond_{PC}^{ST} &= (ICacheMiss + ST_{IR_{1,1}}).\overline{IALU.opcode == BR} \\
cond_{PC}^{BT} &= (IALU.opcode == BR)
\end{aligned}
$$

$$
\begin{aligned}
cond_{IR_{1,1}}^{NF} &= \overline{ST_{PC}}.(\overline{ST_{IR_{2,1}}} + \overline{ST_{IR_{2,2}}} + \overline{ST_{IR_{2,3}}} + \overline{ST_{IR_{2,4}}}) \\
cond_{IR_{1,1}}^{ST} &= ST_{IR_{2,1}}.ST_{IR_{2,2}}.ST_{IR_{2,3}}.ST_{IR_{2,4}} \\
cond_{IR_{1,1}}^{NI} &= ST_{PC}.(\overline{ST_{IR_{2,1}}} + \overline{ST_{IR_{2,2}}} + \overline{ST_{IR_{2,3}}} + \overline{ST_{IR_{2,4}}})
\end{aligned}
$$

$$
\begin{aligned}
cond_{IR_{2,1}}^{NF} &= \overline{ST_{IR_{1,1}}}.\overline{ST_{IR_{9,1}}} \\
cond_{IR_{2,1}}^{ST} &= ST_{IR_{9,1}} \\
cond_{IR_{2,1}}^{NI} &= ST_{IR_{1,1}}.\overline{ST_{IR_{9,1}}}
\end{aligned}
$$

$$
\begin{aligned}
cond_{IR_{2,2}}^{NF} &= \overline{ST_{IR_{1,1}}}.\overline{ST_{IR_{3,1}}} \\
cond_{IR_{2,2}}^{ST} &= ST_{IR_{3,1}} \\
cond_{IR_{2,2}}^{NI} &= ST_{IR_{1,1}}.\overline{ST_{IR_{3,1}}}
\end{aligned}
$$

$$
\begin{aligned}
cond_{IR_{2,3}}^{NF} &= \overline{ST_{IR_{1,1}}}.\overline{ST_{IR_{3,2}}} \\
cond_{IR_{2,3}}^{ST} &= ST_{IR_{3,2}} \\
cond_{IR_{2,3}}^{NI} &= ST_{IR_{1,1}}.\overline{ST_{IR_{3,2}}}
\end{aligned}
$$

$$
\begin{aligned}
cond_{IR_{2,4}}^{NF} &= \overline{ST_{IR_{1,1}}}.\overline{ST_{IR_{9,1}}}.\overline{busy} \\
cond_{IR_{2,4}}^{ST} &= ST_{IR_{9,1}} + busy \\
cond_{IR_{2,4}}^{NI} &= ST_{IR_{1,1}}.\overline{ST_{IR_{9,1}}}.\overline{busy}
\end{aligned}
$$

/* *busy* bit is used to model this multi-cycle unit */

$$
\begin{aligned}
cond_{IR_{3,1}}^{NF} &= \overline{ST_{IR_{2,2}}}.\overline{ST_{IR_{4,1}}} \\
cond_{IR_{3,1}}^{ST} &= ST_{IR_{4,1}} \\
cond_{IR_{3,1}}^{NI} &= ST_{IR_{2,2}}.\overline{ST_{IR_{4,1}}}
\end{aligned}
$$

$$
\begin{aligned}
cond_{IR_{3,1}}^{NF} &= \overline{ST_{IR_{2,2}}}.\overline{ST_{IR_{4,1}}} \\
cond_{IR_{3,1}}^{ST} &= ST_{IR_{4,1}} \\
cond_{IR_{3,1}}^{NI} &= ST_{IR_{2,2}}.\overline{ST_{IR_{4,1}}}
\end{aligned}
$$

$$
\begin{aligned}
cond_{IR_{4,1}}^{NF} &= \overline{ST_{IR_{3,1}}}.\overline{ST_{IR_{5,1}}} \\
cond_{IR_{4,1}}^{ST} &= ST_{IR_{5,1}} \\
cond_{IR_{4,1}}^{NI} &= ST_{IR_{3,1}}.\overline{ST_{IR_{5,1}}}
\end{aligned}
$$

$$
\begin{aligned}
cond_{IR_{5,1}}^{NF} &= \overline{ST_{IR_{4,1}}}.\overline{ST_{IR_{6,1}}} \\
cond_{IR_{5,1}}^{ST} &= ST_{IR_{6,1}}
\end{aligned}
$$

$$cond^{NI}_{IR_{5,1}} = ST_{IR_{4,1}}.\overline{ST_{IR_{6,1}}}$$

$$cond^{NF}_{IR_{6,1}} = \overline{ST_{IR_{5,1}}}.\overline{ST_{IR_{7,1}}}$$
$$cond^{ST}_{IR_{6,1}} = ST_{IR_{7,1}}$$
$$cond^{NI}_{IR_{6,1}} = ST_{IR_{5,1}}.\overline{ST_{IR_{7,1}}}$$

$$cond^{NF}_{IR_{7,1}} = \overline{ST_{IR_{6,1}}}.\overline{ST_{IR_{8,1}}}$$
$$cond^{ST}_{IR_{7,1}} = ST_{IR_{8,1}}$$
$$cond^{NI}_{IR_{7,1}} = ST_{IR_{6,1}}.\overline{ST_{IR_{8,1}}}$$

$$cond^{NF}_{IR_{8,1}} = \overline{ST_{IR_{7,1}}}.\overline{ST_{IR_{9,1}}}$$
$$cond^{ST}_{IR_{8,1}} = ST_{IR_{9,1}}$$
$$cond^{NI}_{IR_{8,1}} = ST_{IR_{7,1}}.\overline{ST_{IR_{9,1}}}$$

$$cond^{NF}_{IR_{3,2}} = \overline{ST_{IR_{2,3}}}.\overline{ST_{IR_{4,2}}}$$
$$cond^{ST}_{IR_{3,2}} = ST_{IR_{4,2}}$$
$$cond^{NI}_{IR_{3,2}} = ST_{IR_{2,3}}.\overline{ST_{IR_{4,2}}}$$

$$cond^{NF}_{IR_{4,2}} = \overline{ST_{IR_{3,2}}}.\overline{ST_{IR_{5,2}}}$$
$$cond^{ST}_{IR_{4,2}} = ST_{IR_{5,2}}$$
$$cond^{NI}_{IR_{4,2}} = ST_{IR_{3,2}}.\overline{ST_{IR_{5,2}}}$$

$$cond^{NF}_{IR_{5,2}} = \overline{ST_{IR_{4,2}}}.\overline{ST_{IR_{9,1}}}$$
$$cond^{ST}_{IR_{5,2}} = ST_{IR_{9,1}}$$
$$cond^{NI}_{IR_{5,2}} = ST_{IR_{4,2}}.\overline{ST_{IR_{9,1}}}$$

$$cond^{NF}_{IR_{9,1}} = (\overline{ST_{IR_{2,1}}} + \overline{ST_{IR_{8,1}}} + \overline{ST_{IR_{5,2}}} + \overline{ST_{IR_{2,4}}}).\overline{ST_{IR_{10,1}}}$$
$$cond^{ST}_{IR_{9,1}} = ST_{IR_{10,1}}$$
$$cond^{NI}_{IR_{9,1}} = (ST_{IR_{2,1}}.ST_{IR_{8,1}}.ST_{IR_{5,2}}.ST_{IR_{2,4}}).\overline{ST_{IR_{10,1}}}$$

$$cond^{NF}_{IR_{10,1}} = \overline{DCacheMiss} + \overline{IR_{10,1}.opcode == LD}$$
$$cond^{ST}_{IR_{9,1}} = DCacheMiss.(IR_{10,1}.opcode == LD)$$
$$cond^{NI}_{IR_{9,1}} = FALSE$$

The property checking is done automatically and returned successful for this modeling. We show here a small trace of the property checking involving $IR_{1,1}$ and $IR_{2,4}$ which demonstrates the simplicity and elegance of the underlying model. We first show that the determinism property is satisfied for $IR_{1,1}$. Using modeling above and Equation (12):

$cond_{IR_{1.1}}^{NF} + cond_{IR_{1.1}}^{ST} + cond_{IR_{1.1}}^{NI}$
$= \overline{ST_{PC}}.(\overline{ST_{IR_{2,1}}} + \overline{ST_{IR_{2,2}}} + \overline{ST_{IR_{2,3}}} + \overline{ST_{IR_{2,4}}}) + ST_{IR_{2,1}}.ST_{IR_{2,2}}.ST_{IR_{2,3}}.ST_{IR_{2,4}} + ST_{PC}.(\overline{ST_{IR_{2,1}}} + \overline{ST_{IR_{2,2}}} + \overline{ST_{IR_{2,3}}} + \overline{ST_{IR_{2,4}}})$
$= (\overline{ST_{IR_{2,1}}} + \overline{ST_{IR_{2,2}}} + \overline{ST_{IR_{2,3}}} + \overline{ST_{IR_{2,4}}}).(\overline{ST_{PC}} + ST_{PC}) + ST_{IR_{2,1}}.ST_{IR_{2,2}}.ST_{IR_{2,3}}.ST_{IR_{2,4}}$
$= (\overline{ST_{IR_{2,1}}} + \overline{ST_{IR_{2,2}}} + \overline{ST_{IR_{2,3}}} + \overline{ST_{IR_{2,4}}}) + ST_{IR_{2,1}}.ST_{IR_{2,2}}.ST_{IR_{2,3}}.ST_{IR_{2,4}}$
$= 1$

In this manner, we can show that $cond_{IR_{1.1}}^{NF}.cond_{IR_{1.1}}^{ST}$ is 0, $cond_{IR_{1.1}}^{NF}.cond_{IR_{1.1}}^{NI}$ is 0, and $cond_{IR_{1.1}}^{ST}.cond_{IR_{1.1}}^{NI}$ is 0. This verifies the determinism property for $IR_{1,1}$. The similar checks are done for all the nodes.

Similarly, for the verification of in-order execution between any two adjacent nodes in the DLX pipeline the properties (Equation (15) - Equation (26)) must be satisfied. For example, $cond_{IR_{1,1}}^{ST}.cond_{IR_{2,4}}^{NF}$ returns 0. In our property checker the verification was successful for all the nodes.

During design space exploration we added a feedback path from $IR_{9,1}$ to $IR_{2,3}$ to see the impact of data forwarding on multiply followed by accumulate intensive benchmarks (e.g., wavelet, low-pass). We modified ADL accordingly by treating $IR_{9,1}$ as one of $IR_{2,3}$'s parent (other than $IR_{1,1}$) and $IR_{2,3}$ as one of $IR_{9,1}$'s children (other than $IR_{10,1}$) and generated necessary conditions. The property checking failed for in-order execution as well as finiteness. A careful observation shows that the second specification ($IR_{2,3}$ as one of $IR_{9,1}$'s children) was wrong since producer unit never waits for the receiver unit to receive the data in this scenario. After removing the second specification the verification is successful. In such a simple situation this kind of specification mistakes might appear as trivial, but when architecture gets complicated and DSE iterations and varieties increases, the potential for introducing bugs also increases.

8 Summary

This report proposed an ADL driven verification of in-order execution in pipelined processors. It uses an FSM-based modeling of pipelined controllers with a special focus on next state functions. Based on the modeling we presented a set of properties which are used to verify the correctness of in-order execution in the pipeline. If a given pipelined processor satisfies all the properties, its pipeline behavior is guaranteed to be correct. We presented an automatic ADL driven verification framework. We used DLX processor to demonstrate the usefulness of our approach.

We are extending our modeling and verification techniques towards VLIW and superscalar processors.

9 Acknowledgments

This work was partially supported by grants from NSF (MIP-9708067), DARPA (F33615-00-C-1632), JSPS postdoctoral fellowship and Motorola Inc. We would like to gratefully acknowledge all EXPRESSION team members for their contribution to the processor verification work.

References

[1] ARC Cores. *http://www.arccores.com.*

[2] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, 1994.

[3] G. G. et al. CHESS: Retargetable code generation for embedded DSP processors. In *Code Generation for Embedded Processors.* Kluwer, 1997.

[4] G. H. et al. ISDL: An instruction set description language for retargetability. In *Proc. DAC*, 1997.

[5] R. L. et al. Retargetable generation of code selectors from HDL processor models. In *Proc. EDTC*, 1997.

[6] M. Freericks. The nML machine description formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Dept., 1993.

[7] P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt, and A. Nicolau. EXPRESSION: An ADL for system level design exploration. Technical Report TR 98-29, University Of California, Irvine, 1998.

[8] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. DATE*, Mar. 1999.

[9] J. Hauke and J. Hayes. Specification and verification of pipelining in the arm2 risc microprocessor. In *ACM TODAES*, volume 3, pages 563–580, October 1998.

[10] J. Hauke and J. Hayes. Microprocessor design verification using reverse engineering. In *HLDVT*, pages 2–9, 1999.

[11] J. Hennessy and D. Patterson. *Computer Architecture: A quantitative approach*. Morgan Kaufmann Publishers Inc, San Mateo, CA, 1990.

[12] R. Ho, C. Yang, M. A. Horowitz, and D. Dill. Architecture validation for processors. In *ISCA*, 1995.

[13] H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose. Automatic test pattern generation for pipelined processors. In *ICCAD*, pages 580–583, 1994.

[14] J. Levitt and K. Olukotun. Verifying correct pipeline implementation for microprocessors. In *ICCAD*, pages 162–169, 1997.

[15] V. Rajesh and R. Moona. Processor modeling for hardware software codesign. In *International Conference on VLSI Design*, Jan. 1999.

[16] J. Skakkebaek, R. Jones, and D. Dill. Formal verification of out-of-order execution using incremental flushing. In *CAV*, 1998.

[17] Tensilica Incorporated. *http://www.tensilica.com.*

[18] H. Tomiyama, T. Yoshino, and N. Dutt. Verification of in-order execution in pipelined processors. In *HLDVT Workshop*, 2000.

[19] Trimaran Release: http://www.trimaran.org. *The MDES User Manual*, 1997.

[20] S. Ur and Y. Yadin. Micro architecture coverage directed generation of test programs. In *DAC*, pages 175–180, 1999.

[21] M. Velev and R. Bryant. Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction. In *DAC*, pages 112–117, 2000.