Automatic Verification of In-Order Execution in Microprocessors with Fragmented Pipelines and Multicycle Functional Units [*]

Prabhat Mishra[†] Hiroyuki Tomiyama[‡] Nikil Dutt[†] Alex Nicolau[†]

[†]Center for Embedded Computer Systems
University of California, Irvine, CA 92697
{pmishra, dutt, nicolau}@cecs.uci.edu

[‡]Institute of Systems & Information Tech.
Fukuoka 814-0001, Japan
tomiyama@isit.or.jp

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 Finite State Machine (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 presence of fragmented pipelines and multicycle functional units. Our approach leverages the system architect's knowledge about the behavior of the pipelined processor, through Architecture Description Language (ADL) constructs, and thus allows a powerful top-down approach to pipeline verification. We applied this methodology to the DLX processor to demonstrate the usefulness of our approach.

1 Introduction

One of the most important problems in today's microprocessor design verification is the lack of a golden reference model that can be used for verifying the design at different levels of abstraction. Current processor design methods typically exhibit a gap between the processor architect's view of the design (reference manuals written in English) and the RT-level implementation (written in HDL) of the design. Thus many existing validation techniques ([4], [8]) 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. A key challenge in today's design verification is to extract the information from the RT-level description of the design and to perform equivalence checking with the model extracted from the given specification (written in English language). Our verification technique is complimentary to these bottom up approaches. Our approach leverages the system architects knowledge about the behavior of the pipelined processor, through Architecture Description Language (ADL) constructs, and thus allows a powerful top-down approach to pipeline verification. The ADL description can serve as a reference model. However, it is necessary to verify the correctness of the ADL specification to ensure that the reference model is well-formed, and that it correctly specifies the desired architectural style and its attendant properties.

The ADL driven validation approach is also a natural choice during rapid Design Space Exploration (DSE) of System-on-Chip (SOC) architectures. Recent approaches on language-driven Design Space Exploration (DSE) ([5], [6]) use 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. 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 DSE may result in incorrect execution of the system and verification techniques can be used to ensure correctness of the modified 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 processors with fragmented pipelines and multicycle functional units. Section 5 proposes our verification technique followed by a case study in Section 6. Section 7 concludes the paper.

2 Related Work

Several approaches for formal or semi-formal verification of pipelined processors has been developed in the past. Theorem proving techniques, for example, have been successfully adapted to verify pipelined processors ([3], [16], [18]). However, these approaches require a great deal of user in-

[*] This work was partially supported by grants from NSF (MIP-9708067), DARPA (F33615-00-C-1632), Motorola Inc., and Hitachi Ltd.

tervention, especially for verifying control intensive designs. 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 [17, 21]. Mark Aagaard [1] presented a design framework for pipelines with structural hazards. He created a framework with four parameters that are used to characterize and verify pipelines. 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. All the above techniques 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. 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 FSM-based model.

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

Tomiyama et al. [19] presented FSM based modeling of pipelined processors with in-order execution and is closest to our approach. Their model can handle only simple processors with straight pipelines. Our work extends this model to handle more realistic processor features such as fragmented pipelines and multicycle functional units. Furthermore, we present an automatic property checking framework driven by an Architecture Description Language (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 description in an ADL. The FSM model of the pipelined processor description is automatically generated from this ADL description. We have defined several properties such as determinism, in-order execution, and finiteness, to ensure that the ADL description of the architecture is well-formed. Our automatic property checking framework determines if all the necessary properties are satisfied or not. In case of 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) can be generated for design space exploration. Furthermore, this ADL specification can be used as a golden reference model for verifying designs at different levels of abstraction (behavior, RTL, gate, transistor etc.).

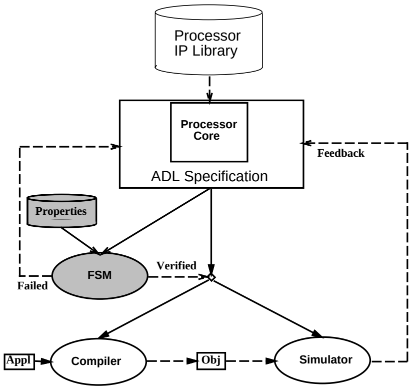

Figure 1. The Flow in our approach

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

Any ADL that can capture both the structure and behavior of the processor can be used in our verification and exploration framework. We have chosen the EXPRESSION ADL [6] that captures the structure, behavior, and the mapping between them for the processor pipeline. However, for the property verification we need to specify the conditions for stalling, normal flow and nop insertion for each functional unit along with the structure, behavior and the mapping information. Figure 2 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 describe how to specify the conditions for stalling, normal flow and nop insertion in the ADL.

A unit can be *stalled* due to external signals or due to contributions arising inside the processor. For example, the external signal that can stall a fetch unit is *ICache_Miss*; internal condition for stalling of fetch unit can be due to decode stall, hazards or exceptions. For units with multiple children the stalling condition due to internal contribution may differ. For example, the unit $UNIT_{i-1,j}$ in Figure 2 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, 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. This is not sufficient for a unit with multiple children (fork node). For example, the fork node, $UNIT_{i-1,j}$ in Figure 2, has q children. Designer need to specify in the ADL the stall condition of the fork node in the presence of different types of instructions. The ADL description has structural information for each of these q pipelines: how many pipeline stages are there in each pipeline, how many cycles needed by an instruction in a particular pipeline stage etc. Designers also need to specify if the fork node needs to be stalled when the current instruction is for the pipeline i ($1 \leq i \leq q$) and there are instructions in pipelines j ($1 \leq j \leq q$).

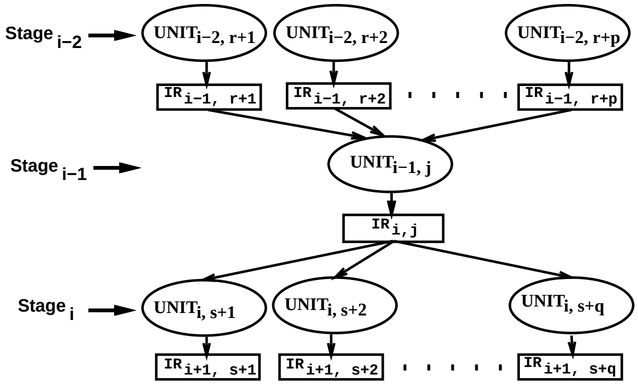

Figure 2. A fragment of the processor pipeline

A unit is in *normal flow* if it can receive instruction from its parent unit and can send to its child unit. 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 be in normal flow depending on several combinations of states among its parent units and child units: 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, designer selects 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 its parent (or busy computing in case of multicycle unit) and its child unit 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: all of its parents are stalled and one of its children is not stalled etc. During specification, designer selects from the set (ANY, SOME, ALL, NONE) the contributions from parents and children to specify the nop insertion condition for each unit.

The Program Counter (PC) unit can be *stalled* 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* state. The PC unit is in *sequential execution* 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. This paper especially focuses on the next state function of the FSM. Figure 3 shows the FSM model of the fragment in Figure 2.

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 pipelines). Each single-cycle pipeline register takes one cycle if there are no pipeline hazards. A 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 an 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 an instruction from instruction memory pointed by program counter PC, and stores the instruction into the first instruction register $IR_{1,j}$ ($1 \leq j \leq N_1$). During execution the instruction stored in $IR_{i,j}$ is executed at $Stage_i$ and then stored into the next instruction register $IR_{i+1,s+k}$ ($1 \leq k \leq q$)

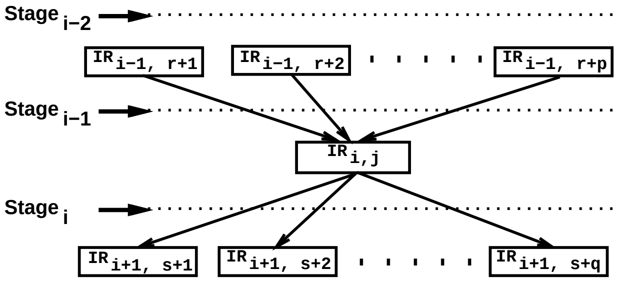

Figure 3. FSM model of the fragment in Figure 2

In this paper, we define a state of the N-stage pipeline as values of PC and $(N-1) \times \sum_{i=1}^{N-1} N_i$ ($= M$ say) instruction registers. 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 describe the conditions for stalling(ST), normal flow(NF), nop insertion(NI), sequential execution(SE), 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 hazards, exceptions, or external interrupts on $IR_{i,j}$ ($ST^{self}_{IR_{i,j}}$ say). However, as mentioned in Section 4.1, $ST^{self}_{IR_{i,j}}$ has another component, which is instruction specific. More formally the condition for stalling at time t in the presence of a set of external signals $I(t)$ on $S(t)$ is, $cond^{ST}_{IR_{i,j}}(S(t),I(t))$ ($cond^{ST}_{IR_{i,j}}$ in short),

$$cond^{ST}_{IR_{i,j}} = ST_{IR_{i,j}} = ST^{child}_{IR_{i,j}} + ST^{self}_{IR_{i,j}} \tag{2}$$

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

$$cond^{ST}_{IR_{i,j}} = ST_{IR_{i,j}} = \cap^{q}_{k=1} ST_{IR_{i+1,s+k}} + ST^{self}_{IR_{i,j}}$$

Let us further assume, the designer has specified that $IR_{i,j}$ (fork node) is stalled if the current instruction is for pipeline k (I_k say) and there are instructions in pipelines x ($x \neq k; 1 \leq x,k \leq q$) which may reach later than current instruction (if issued) to the join node (corresponding to the fork node $IR_{i,j}$). Let us define $\tau(y)$ as the total number clock cycles needed by pipeline y. This definition can be recursively applied for pipelines containing fragmented pipelines. This is derived from the ADL description by using number of pipeline stages in the pipeline y (starting at fork node $IR_{i,j}$ and ending at the corresponding join node) and the timing of each pipeline stage. Then $ST^{self}_{IR_{i,j}}$ for the previous equation becomes

$$ST^{self}_{IR_{i,j}} = \cup^{q}_{k=1}(I_k \;.\; \cup^{q}_{x=1} S_{x,k})$$

where, $S_{x,k}$ is 1 if the latest instruction in the pipeline x is active for less than $(\tau(x) - \tau(k))$ cycles.

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,

$$cond^{NF}_{IR_{i,j}} = NF^{parent}_{IR_{i,j}} \;.\; NF^{child}_{IR_{i,j}} \;.\; \overline{ST^{self}_{IR_{i,j}}} \tag{3}$$

For example, if the designer specified that $IR_{i,j}$ will be in normal flow if ”ANY” (see Section 4.1) of its parents is not stalled and ”ANY” of its 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,

$$cond^{NI}_{IR_{i,j}} = NI^{parent}_{IR_{i,j}} \;.\; NI^{child}_{IR_{i,j}} \;.\; \overline{ST^{self}_{IR_{i,j}}} \tag{4}$$

For example, if the 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., $cond^{SE}_{PC}$ (SE: sequential execution), $cond^{NI}_{PC}$ (NI: nop insertion), and $cond^{BT}_{PC}$ (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,

$$cond^{SE}_{PC}(S(t),I(t)) = NF^{child}_{PC} \;.\; \overline{ST^{self}_{PC}} \;.\; \overline{BT_{PC}} \tag{5}$$

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

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

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., $IR_{i,j}(t+1) = IR_{i,j}(t)$. Note that, in general, any instruction in the pipeline cannot skip pipe stages. For example, $IR_{i,j}(t+1)$ cannot be $IR_{i-2,v}(t)$ ($1 \leq v \leq N_{i-2}$) if there are no feed-forward paths. 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 $(M+1)$ registers ($M = (N-1) \times \sum^{N-1}_{i=1} N_i$). 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^{NS}_{PC} and $f^{NS}_{IR_{i,j}}$ ($1 \leq i \leq N-1$, $1 \leq j \leq N_i$) denote next state functions for PC and $IR_{i,j}$ respectively. Note that in general $f^{NS}_{IR_{i,j}}$ 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.

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

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 $cond^{x}_{PC}$'s ($x \in (SE,BT,ST)$) are logic functions of $S(t)$ and $I(t)$ as described in Equation (5) - Equation (7), and return either 0 or 1. For example, if $cond^{ST}_{PC}(S(t),I(t))$ is 1, PC keeps its current value at the next cycle.

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

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

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

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

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 in instruction memory (IM). 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 $cond^{NI}_{IR_{i,j}}(S(t),I(t))$ is 1, resulting in $IR_{i,j}(t+1) = nop$.

At present, signals coming from the datapath or the memory 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 Verification of In-Order Execution

Based on the FSM modeling presented in Section 4, we propose a method to verify the correctness of pipeline controllers with 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. We first describe the properties needed for verifying the in-order execution, then we present our automatic property checking framework driven by an ADL.

5.1 Properties

This section presents three properties: determinism, in-order execution, and finiteness. Any pipelined processor must satisfy these properties to ensure correct in-order execution.

Determinism

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

$$
\begin{aligned}
&cond^{SE}_{PC}(S(t),I(t)) + cond^{BT}_{PC}(S(t),I(t)) +\\
&cond^{ST}_{PC}(S(t),I(t)) = 1
\end{aligned} \tag{11}
$$

$$
\begin{aligned}
&cond^{NF}_{IR_{i,j}}(S(t),I(t)) + cond^{ST}_{IR_{i,j}}(S(t),I(t)) +\\
&cond^{NI}_{IR_{i,j}}(S(t),I(t)) = 1,\\
&\forall i,j(1 \leq i \leq N-1, 1 \leq j \leq N_i)
\end{aligned} \tag{12}
$$

$$
\begin{aligned}
&cond^{x}_{PC}(S(t),I(t)) \;.\; cond^{y}_{PC}(S(t),I(t)) = 0,\\
&\forall x,y(x,y \in \{SE,BT,ST\} \;\wedge\; x \neq y)
\end{aligned} \tag{13}
$$

$$
\begin{aligned}
&cond^{x}_{IR_{i,j}}(S(t),I(t)) \;.\; cond^{y}_{IR_{i,j}}(S(t),I(t)) = 0,\\
&\forall i,j(1 \leq i \leq N-1, 1 \leq j \leq N_i),\\
&\forall x,y(x,y \in \{NF,ST,NI\} \;\wedge\; x \neq y)
\end{aligned} \tag{14}
$$

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 states $S(t)$ and external signals $I(t)$. The last two guarantee that any two conditions are disjoint for each next state function. Informally, exactly one of the conditions should be true in a cycle for each state register.

In-Order Execution

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 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 are allowed, the instruction stored in $IR_{i-1,r+l}$ ($1 \le l \le p$) 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.

$$\cap_{l=1}^{p} cond^{ST}_{IR_{i-1,r+l}} \,.\, cond^{NF}_{IR_{i,j}} = 0 \qquad (15)$$
$$(2 \le i \le N-1, 1 \le j \le N_i, 1 \le l \le p)$$

- 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.

$$cond^{NF}_{IR_{i,j}} \,.\, \cap_{k=1}^{q} cond^{ST}_{IR_{i+1,s+k}} = 0 \qquad (16)$$
$$(2 \le i \le N-1, 1 \le j \le N_i, 1 \le k \le q)$$

- 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 $IR_{i-1,r+l}$ ($1 \le l \le p$) at time t will be overwritten by the nop instruction. More formally, the Equation (17) should be satisfied.

$$cond^{NI}_{IR_{i,j}} \,.\, \cap_{k=1}^{q} cond^{ST}_{IR_{i+1,s+k}} = 0 \qquad (17)$$
$$(2 \le i \le N-1, 1 \le j \le N_i, 1 \le k \le q)$$

- 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.

$$cond^{NF}_{IR_{i-1,r+l}} \,.\, cond^{NI}_{IR_{i,j}} = 0 \qquad (18)$$
$$(2 \le i \le N-1, 1 \le j \le N_i, 1 \le l \le p)$$

- 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.

$$cond^{NI}_{IR_{i-1,r+l}} \,.\, cond^{NI}_{IR_{i,j}} = 0 \qquad (19)$$
$$(2 \le i \le N-1, 1 \le j \le N_i, 1 \le l \le p)$$

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:

$$\forall (F,J), I_a \preceq_J I_b \Rightarrow \Gamma_F(I_a) < \Gamma_F(I_b) \qquad (20)$$

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):

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

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

$$cond^{ST}_{PC} \,.\, cond^{NF}_{IR_{1,j}} = 0 \qquad (22)$$
$$cond^{SE}_{PC} \,.\, \cap_{j=1}^{N_1} cond^{ST}_{IR_{1,j}} = 0 \qquad (23)$$
$$cond^{BT}_{PC} \,.\, \cap_{j=1}^{N_1} cond^{ST}_{IR_{1,j}} = 0 \qquad (24)$$
$$cond^{SE}_{PC} \,.\, cond^{NI}_{IR_{1,j}} = 0 \qquad (25)$$
$$cond^{BT}_{PC} \,.\, cond^{NI}_{IR_{1,j}} = 0 \qquad (26)$$

All of the above equations (Equation (15) - Equation (26)) must hold to ensure correct in-order execution.

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 indefinitely. 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 $IR_{k,y}$ where $k \ge i$, but cannot be a function of IR_k where $k < i$.

5.2 Automatic Verification Framework

In this section we describe our automatic property checking framework. The first step is to describe the processor pipeline in the EXPRESSION ADL [6]. The finite state machine (FSM) model of the processor controller is automatically extracted from the ADL description. Next, the properties described in Section 5.1 are automatically applied to the FSM model to verify the in-order execution.

The framework first generates the flow equations for NF, ST, and NI for each instruction register (SE, ST, and BT for PC) using ADL description and Equation (1) - Equation (7), then it generates the equations necessary for verifying properties using ADL description and Equation (11) - Equation (26). The *Eqntott* [9] tool converts these equations in two-level representation of a two-valued Boolean function. This two-level representation is fed to *Espresso* [12] 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 details on the verification framework can be found in [15].

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 has few interesting features viz., fragmented pipelines, multicycle units etc. Figure 4 shows the DLX processor pipeline.

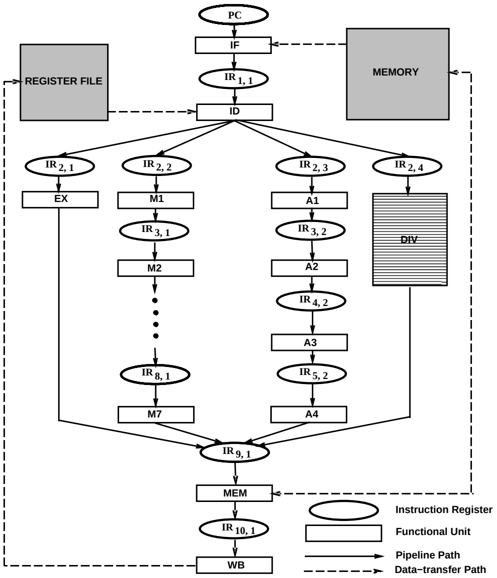

Figure 4. The DLX Processor

We used the EXPRESSION ADL [6] to capture the structure and behavior of the DLX processor. 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 children units for stall condition. While capturing normal flow condition for each unit we selected ”ANY” for parent units and ”ANY” for children units. Similarly, for each unit we specified ”ALL” as contribution from parent units and ”ANY” as contribution for children units for nop insertion. For example, the condition specification for the decode unit (no self contribution) is shown below.

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

Using the ADL description, we automatically generated the equations for flow conditions for all the units [15]. For example, the equation for the stall condition for the decode latch is shown below (using Equation (2), and the description of the decode unit shown above).

$$cond^{ST}_{IR_{1,1}} = \overline{ST_{IR_{2,1}}} \;.\; \overline{ST_{IR_{2,2}}} \;.\; \overline{ST_{IR_{2,3}}} \;.\; \overline{ST_{IR_{2,4}}} \qquad (27)$$

$IR_{2,4}$ 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 $ST^{self}_{IR_{2.4}}$ as shown in Equation (2).

The necessary equations for verifying the properties viz., determinism, in-order execution etc., are generated automatically from the given ADL description. We show here a small trace of the property checking to demonstrate the simplicity and elegance of the underlying model. We show that the determinism property is satisfied for $IR_{1,1}$ using the modeling above:

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

We have used *Espresso* to minimize the equations. These minimized equations are analyzed to verify whether the properties are violated or not. The complete verification took 41 seconds on a 333 MHz Sun Ultra-5 with 128M RAM. Our framework determined that the Equation (20) 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 n to floating-point adder pipeline (A1 - A4) and an integer addition operation I_{iadd} to integer ALU (EX) at cycle *n+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 processor model and the properties. The only difference is $ST^{self}_{IR_{i,j}}$ for decode unit

(Equation (2)) becomes: $ST^{self}_{IR_{i,j}} =$

$I_1 \, . \, (S_{2,1} + S_{3,1} + S_{4,1}) + I_2 \, . \, S_{4,2} + I_3 \, . \, (S_{2,3} + S_{4,3})$

where, the numbers 1, 2, 3, and 4 are the pipelines (between ID and MEM unit) integer ALU, MUL, FADD, and DIV respectively. The signal $S_{x,y}$ is 1 if the latest instruction in pipeline x is active for less than $(\tau(x) - \tau(y))$ cycles. Here, $\tau(x)$ returns the total number of clock cycles needed by pipeline x ($\tau(1) = 1$,$\tau(2) = 7$, $\tau(3) = 4$, $\tau(4) = 25$). The instructions I_1, I_2, I_3, and I_4 represent the instructions supported by the pipelines 1, 2, 3, and 4 respectively. Informally, this equation means that if current instruction is I_2 (multiply) and there is a instruction in DIV unit which is active for less than 18 cycles ($\tau(4) - \tau(2)$ = 25 - 7 = 18) etc. then decode should stall. Otherwise, it leads to out-of-order execution. Note that, the equation does not have any term for I_4. This is because $S_{x,4}$ can never be 1 since $(\tau(x) - \tau(4))$ is always negative. For the same reason, all the terms in the equation does not have four $S_{x,y}$.

The Equation (21) is violated for this modeling for $IR_{9,1}$. The instruction sequence generated by our framework for this failure consists of a multiply operation (issued by decode unit in cycle n) followed by a floating-point add operation (issued by decode unit in cycle *(n + 3)*). As a result both the operations reach $IR_{9,1}$ at cycle *(n+7)*.

We modified the ADL description to redefine $S_{x,y}$ signal: it is 1 if the latest instruction in pipeline x is active for less than *or equal to* $(\tau(x) - \tau(y))$ cycles. The in-order execution was successful for this modeling.

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 and lowpass from multimedia and DSP domains). We modified the 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 the producer unit never waits for the receiver unit to receive the data in this scenario. After removing the second specification the verification was successful. In such a simple situation this kind of specification mistakes might appear as trivial, but when the architecture gets complicated and DSE iterations and varieties increases, the potential for introducing bugs also increases.

7 Summary

This paper proposed an ADL driven verification of in-order execution in processors with fragmented pipelines and multicycle functional units. 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 the in-order execution of the pipeline. We presented an automatic ADL driven verification framework. We used the DLX processor as an example to demonstrate the usefulness of our approach.

Currently, our verification approach uses equation based property checking framework. We are in the process of using existing model checkers, such as SMV [10] and VIS [11], to perform property checking.

We are extending our modeling and verification technique towards VLIW and superscalar processors. We are also studying how to extend this model to support multiple interrupts and exceptions.

References

[1] M. Aagaard. *A Design Framework for Pipelines With Structural Hazards*. Dept. of ECE, University of Waterloo, 2001.

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

[3] D. Cyrluk. Microprocessor verification in pvs: A methodology and simple example. Technical report, SRI-CSL-93-12, 1993.

[4] P. Ho et al. Formal verification of pipeline control using controlled token nets and abstract interpretation. In *ICCAD*, 1998.

[5] V. Zivojnovic et al. LISA - machine description language and generic machine model for HW/SW co-design. In *IEEE Workshop on VLSI Signal Processing*, 1996.

[6] A. Halambi et al. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. *DATE*, 1999.

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

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

[9] http://buffy.eecs.berkeley.edu. *Berkeley PLA Tools*.

[10] http://www.cs.cmu.edu/ modelcheck *Symbolic Model Verifier*.

[11] http://www-cad.eecs.berkeley.edu/Respep/Research/vis *VIS Home*.

[12] http://www-cad.eecs.berkeley.edu. *Espresso*.

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

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

[15] P. Mishra et al. Architecture description language driven verification of in-order execution in pipelined processors. Technical Report 01-20, University of California, Irvine, 2001.

[16] J. Sawada and J. W.A. Hunt. Trace table based approach for pipelined microprocessor verification. In *CAV*, 1997.

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

[18] M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. In *IEEE Software*, volume 7(5), pages 52–64, 1990.

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

[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.