# Architecture Description Language driven Validation of Dynamic Behavior in Pipelined Processor Specifications

Prabhat Mishra
pmishra@cecs.uci.edu

Nikil Dutt
dutt@cecs.uci.edu

Center for Embedded Computer Systems
University of California, Irvine, CA 92697, USA

Hiroyuki Tomiyama
tomiyama@is.nagoya-u.ac.jp

Dept. of Information Engineering
Nagoya University, Nagoya 464-8603, Japan

CECS Technical Report #03-25
Center for Embedded Computer Systems
University of California, Irvine, CA 92697, USA

*July 28, 2003*

## Abstract

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

## Contents

## List of Figures

## List of Tables

## 1 Introduction

Traditional embedded systems consist of programmable processors, coprocessors, application specific integrated circuits (ASIC), memories, and I/O (input/output) interfaces. Figure 1 shows a traditional hardware/software co-design flow. The embedded system is specified in a system design language. The specification is then partitioned into tasks that are either assigned to software (i.e., executed on the processor), or hardware (ASIC) based on design constraints (cost power, and performance). Tasks assigned to hardware are translated into HDL (Hardware Description Language) descriptions and then synthesized into ASICs. The tasks assigned to software are translated into programs (either in high level languages such as C/C++ or in assembly), and then compiled into object code that resides in instruction memory of the processor.

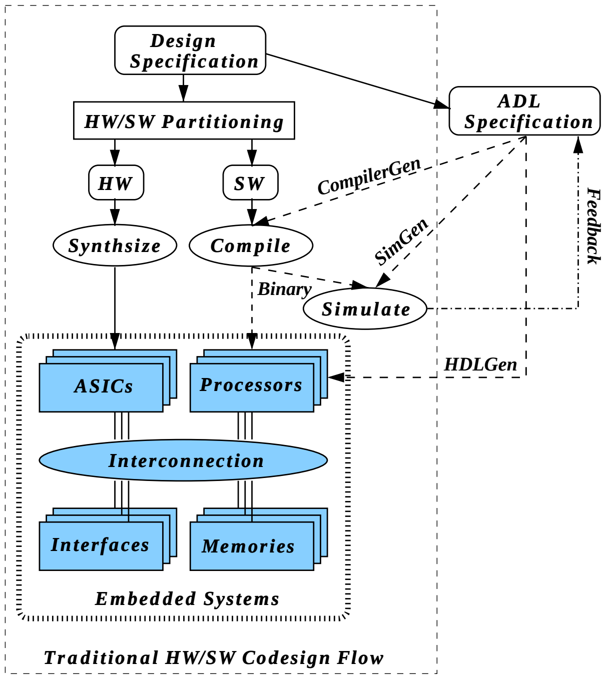

**Figure 1. Hardware/Software Co-Design flow for Embedded Systems**

The traditional HW/SW codesign flow assumes that the embedded system uses an off-the-shelf processor core that has the software toolkit (including compiler and simulator) available. If the processor is not available, the software toolkit needs to be developed for the intended processor. This is a time-consuming process. Moreover, during early design space exploration (DSE), system designers would like to make modifications to programmable architecture (processor, coprocessor, memory subsystem) to meet diverse requirements such as low power, better performance, smaller area, and higher code density. Early time-to-market pressure coupled with short product lifetimes make the manual software toolkit generation for each exploration practically infeasible.

The architecture description language (ADL) based codesign flow (shown in Figure 1) solves this problem. The programmable architecture (processor, coprocessor, memory subsystem) of the embedded system is specified in an ADL and the software toolkit can be generated automatically from this description. During early design space exploration the system designer modifies the programmable architecture by modifying the ADL specification to exploit the application (software tasks) behavior. The software toolkit is generated automatically that reflects the changes of the architecture modifications. It is also possible to synthesize the processor core from the ADL specification.

The ADL driven design space exploration has been addressed extensively in both academia and industry. However, the validation of the ADL specification has not been addressed so far. It is important to validate the ADL description of the architecture to ensure the correctness of both the architecture specified, as well as the generated software toolkit. The benefits of validation are 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 exploration may result in incorrect execution of the system and verification techniques can be used to ensure correctness of the modified architecture.

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

In our ADL driven exploration flow, the designer describes the processor architecture in EXPRESSION ADL [1]. To ensure that the ADL specification describes a well-formed architecture we verify both static and dynamic behavior of the processor specification. Earlier, we have developed validation techniques to ensure that the static behavior of the pipeline is well-formed by analyzing the structural aspects of the specification using a graph based model [32]. In this paper, we verify the dynamic behavior of the processor's description by analyzing the instruction flow in the pipeline using a Finite State Machine (FSM) based model to validate several important architectural properties such as determinism, finiteness, and execution style (e.g., in-order execution) in the presence of hazards and multiple exceptions. Our automatic property checking framework determines if all the necessary properties are satisfied or not. In case of a failure, it generates traces so that designer can modify the ADL specification of the architecture. We applied this methodology to a single-issue DLX processor to demonstrate the usefulness of our approach.

The rest of the paper is organized as follows. Section 2 and Section 3 presents related work addressing architecture description languages and validation of pipelined processors. Section 4 outlines our validation approach and the overall flow of our environment. Section 5 presents our FSM based modeling of processor pipelines. Section 6 proposes our validation technique followed by a case study in Section 7. Section 8 concludes the paper.

## 2 Architecture Description Languages

Traditionally, ADLs have been classified into two categories depending on whether they primarily capture the behavior (Instruction-Set) or the structure of the processor. Recently, many ADLs have been proposed that captures both the structure and the behavior of the architecture.

**nML** [23] and **ISDL** [8] are examples of behavior-centric ADLs. In nML, the processor's instruction-set is described as an attributed grammar with the derivations reflecting the set of legal instructions. The nML has been used by the retargetable code generation environment CHESS [7] to describe DSP and ASIP processors. In ISDL, constraints on parallelism are explicitly specified through illegal operation groupings. This could be tedious for complex architectures like DSPs which permit operation parallelism (e.g. Motorola 56K) and VLIW machines with distributed register files (e.g. TI C6X). The retargetable compiler system by Yasuura et al.[9] produces code for RISC architectures starting from an instruction set processor description, and an application described in **Valen-C**. Valen-C is a C language extension supporting explicit and exact bit-width for integer type declarations, targeting embedded software. The processor description represents the instruction set, but does not appear to capture resource conflicts, and timing information for pipelining.

**MIMOLA** [35] is an example of an ADL which primarily captures the structure of the processor wherein the net-list of the target processor is described in a HDL like language. One advantage of this approach is that the same description is used for both processor synthesis, and code generation. The target processor has a micro-code architecture. The net-list description is used to extract the instruction set [36], and produce the code generator. Extracting the instruction set from the structure may be difficult for complicated instructions, and may lead to poor quality code. The MIMOLA descriptions are generally very low-level, and laborious to write.

More recently, languages which capture both the structure and the behavior of the processor, as well as detailed pipeline information (typically specified using Reservation Tables) have been proposed. **LISA** [45] is one such ADL whose main characteristic is the operation-level description of the pipeline. **RADL** [5] is an extension of the LISA approach that focuses on explicit support of detailed pipeline behavior to enable generation of production quality cycle- and phase-accurate simulators. **FLEXWARE** [33] and **MDes** [43] have a mixed-level structural/behavioral representation. FLEXWARE contains the CODESYN code-generator and the Insulin simulator for ASIPs. The simulator uses a VHDL model of a generic parameterizable machine. The application is translated from the user-defined target instruction set to the instruction set of this generic machine. The **MDes** [43] language used in the Trimaran system is a mixed-level ADL, intended for DSE. Information is broken down into sections (such as format, resource-usage, latency, operation, register etc.), based on a high-level classification of the information being represented. However, MDes allows only a restricted retargetability of the simulator to the HPL-PD processor family. MDes permits the description of the memory system, but is limited to the traditional hierarchy (register files, caches, etc.).

The **EXPRESSION** ADL [1] follows a mixed-level approach (behavioral and structural) to facilitate automatic software toolkit generation, validation, HDL generation, and design space exploration for a wide range of programmable embedded systems. The ADL captures the structure, behavior, and mapping (between structure and behavior) of the architecture as shown in Figure 2.

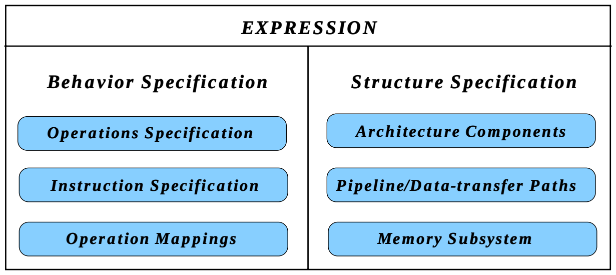

**Figure 2. The EXPRESSION ADL**

The ADL captures all the architectural components and their connectivity as a netlist. It considers four types of components: units (e.g., ALUs), storages (e.g., register files), ports, and connections (e.g., buses). It captures two types edges in the netlist: pipeline edges and data transfer edges. The pipeline edges specify instruction transfer between units via pipeline latches, whereas data transfer edges specify data transfer between components, typically between units and storages or between two storages. It has the ability to capture novel memory subsystem. The behavior is organized into operation groups, with each group containing a set of operations having some common characteristics. Each operation is then described in terms of it's opcode, operands, and behavior. The mapping functions map components in the structure to operations in the behavior. It defines, for each functional unit, the set of operations supported by that unit (and vice versa).

## 3 Related Work

An extensive body of recent work addresses architectural description language driven software toolkit generation and design space exploration for processor-based embedded systems, in both academia: ISDL [8], Valen-C [9], MIMOLA [35], LISA [45], nML [23], [44], and industry: ARC [2], Axys [3], RADL [5], Target [39], Tensilica [40], MDES [43]. However, none of these approaches address the validation issue of the ADL specification. The validation is necessary to ensure the correctness of the generated software toolkit. It is important to ensure that the reference model (specification) is golden, and it describes a well-formed architecture with intended execution style.

Several approaches for formal or semi-formal verification of pipelined processors have been developed in the past. Theorem proving techniques, for example, have been successfully adapted to verify pipelined processors ([6], [19], [20], [25]). However, these approaches require a great deal of user intervention, especially for verifying control intensive designs. Hosabettu [37] proposed an approach to decompose and incrementally build the proof of correctness of pipelined microprocessors by constructing the abstraction function using completion functions.

Burch and Dill presented a technique for formally verifying pipelined processor control circuitry [15]. 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 [21, 24, 26]. The approach of Velev and Bryant [26] focuses on efficiently checking the commutative condition for complex microarchitectures by reducing the problem to checking equivalence of two terms in a logic with equality, and uninterpreted function symbols.

Huggins and Campenhout verified the ARM2 pipelined processor using Abstract State Machine [16]. In [18], 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. A framework for microprocessor correctness statements about safety that is independent of implementation representation and verification approach is presented in [22].

Ho et al. [27] extract controlled token nets from a logic design to perform efficient model checking. Jacobi [4] used a methodology to verify out-of-order pipelines by combining model checking for the verification of the pipeline control, and theorem proving for the verification of the pipeline functionality. Compositional model checking is used to verify a processor microarchitecture containing most of the features of a modern microprocessor [34].

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. Our validation technique is complementary to these formal approaches. We define a set of properties which have to be satisfied for the correct pipeline behavior, and verify the correctness of pipelined processor specifications by applying these properties using a FSM-based model.

## 4 Our Validation Approach

Figure 3 shows our ADL driven exploration flow. The designer describes the processor architecture in EXPRESSION ADL [1]. It is necessary to validate the ADL specification to ensure the correctness of the generated software toolkit and the HDL implementation. To ensure that the ADL specification describes a well-formed architecture we verify both static and dynamic behavior of the processor specification.

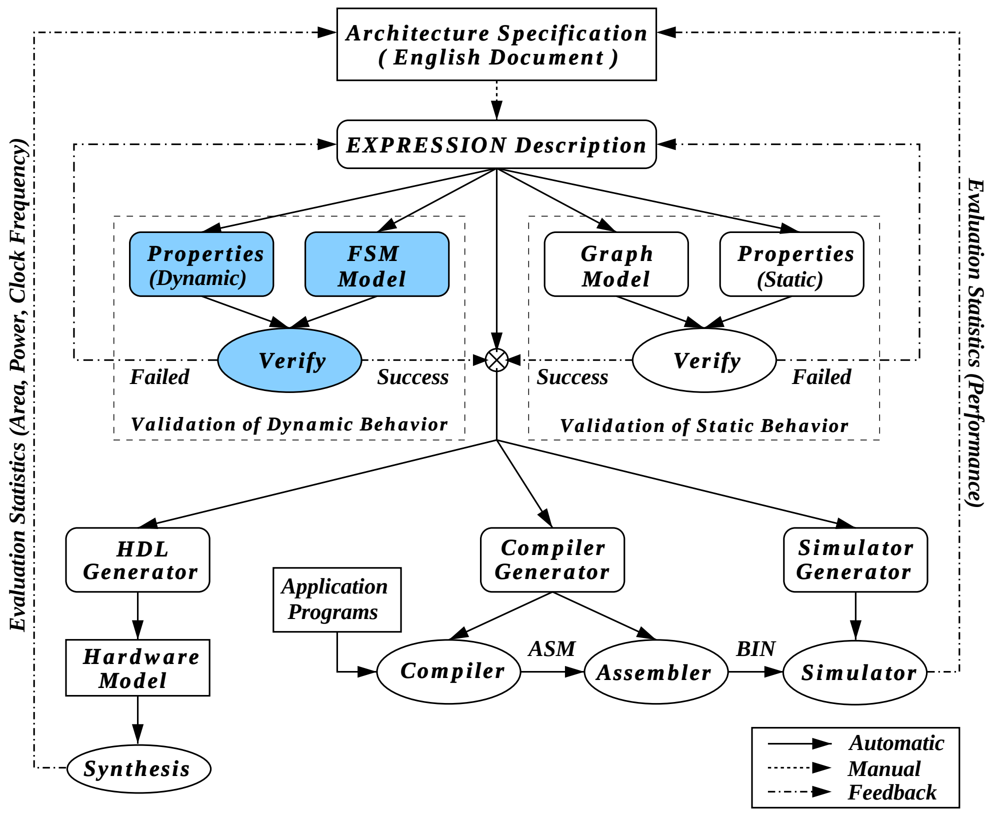

**Figure 3. ADL driven Validation and Exploration Flow**

We have developed validation techniques to ensure that the static behavior of the pipeline is well-formed by analyzing the structural aspects of the specification using several properties, such as connectedness of components, false pipeline and data-transfer paths, operation completeness, and finiteness of execution. These properties are applied on the graph model of the processor specification to verify the static behavior of the specification [32].

We verify the dynamic behavior of the processor by analyzing the instruction flow in the pipeline using a Finite State Machine (FSM) based model to validate several important architectural properties such as determinism, finiteness, and execution style (e.g., in-order execution) in the presence of hazards and multiple exceptions.

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

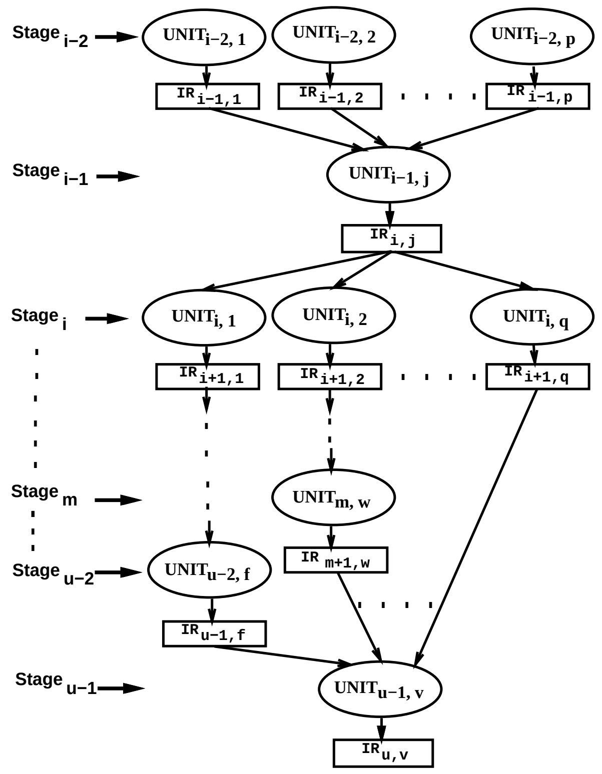

**Figure 4. A fragment of the processor pipeline**

feedback is used by the designer to make modifications to the specification to obtain the best architecture possible for the given set of application programs and design constraints.

## 5 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.

### 5.1 Processor Pipeline Description in ADL

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

The structure (of a processor) is defined by its components (units, storages, ports, connections) and the connectivity (pipeline and data-transfer paths) between these components. Each component is defined by its attributes: the list of opcodes it supports, execution timing for each supported opcode etc. The behavior of a processor is defined by its instruction set. Each operation in the instruction-set is defined in terms of opcode, operands and the functionality of the operation. Figure 4 shows a fragment of a processor pipeline. The oval boxes represent units, rectangular boxes represent pipeline latches, and arrows represent pipeline edges. The detailed description of how to specify structure, behavior, hazards, stalls, interrupts, and exceptions is available in [31].

In this section we briefly describe how we specify conditions for stalling, normal flow, bubble insertion, exception and squashing in ADL. The detailed description is available in [28].

The flow conditions for units are shown in Figure 5. A unit is in *normal flow* (NF) if it can receive instruction from its parent unit and can send to its child unit. A unit can be *stalled* (ST) due to external signals or due to conditions arising inside the processor pipeline. For example, the external signal that can stall a fetch unit is *ICache_Miss*; the internal conditions to stall the fetch unit can be due to decode stall, hazards, or exceptions. A unit performs *bubble insertion* (BI) when it does not receive any instruction from its parent (or busy computing in case of multicycle unit) and its child unit is not stalled. A unit can be in *exception* condition due to internal contribution or due to an exception. A unit is in bubble/nop *squashed* (SQ) stage when it has nop instruction that gets removed or overwritten by an instruction of the parent unit.

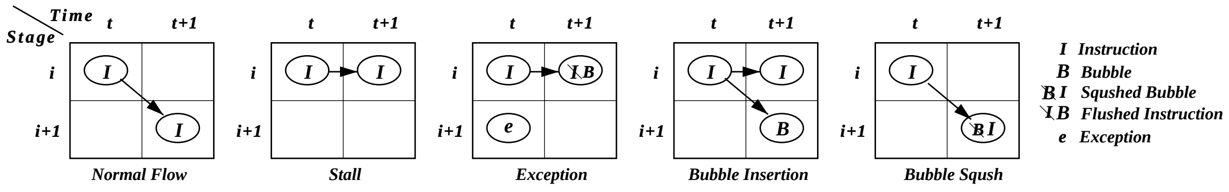

**Figure 5. Flow Conditions for Pipeline Latches**

For units, with multiple children the flow conditions due to internal contribution may differ. For example, the unit $UNIT_{i-1,j}$ in Figure 4 with $q$ children can be *stalled* when *any* one of its children is stalled, or when *some* of its children are stalled (designer identifies the specific ones), or when *all* of its children are stalled; or when *none* of its children are stalled. During specification, the designer selects from the set *(ANY, SOME, ALL, NONE)* the internal contribution along with any external signals to specify the stall condition for each unit. Similarly, the designer specifies the internal contribution for other flow conditions [28].

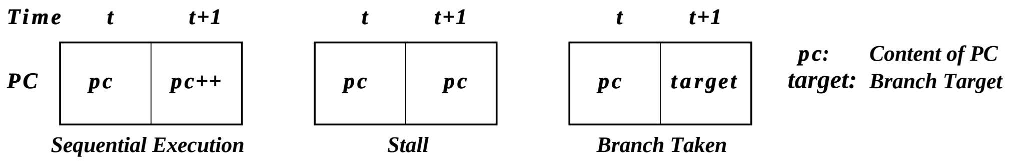

**Figure 6. Flow Conditions for Program Counter (PC)**

The flow conditions for Program Counter (PC) is shown in Figure 6. The PC unit can be *stalled* (ST) due to external signals such as cache miss or when the fetch unit is stalled. When a branch is taken the PC unit is said to be in *branch taken* (BT) state. The PC unit is in *sequential execution* (SE) mode when the fetch unit is in normal flow, there are no external interrupts, and the current instruction is not a branch instruction.

## 5.2 FSM Model of Processor Pipelines

This section presents an FSM-based modeling of controllers in pipelined processors. Figure 7 shows a fragment of the processor pipeline with only instruction registers. We assume a pipelined processor with in-order execution as the target for modeling and validation. The pipeline consists of $n$ stages. Each stage can have more than one pipeline registers (in case of fragmented pipelines). Each single-cycle pipeline register takes one cycle if there are no pipeline hazards. A multi-cycle pipeline register takes $m$ cycles during normal execution (no hazards). In this paper we call these pipeline registers *instruction registers*

(IR) since they are used to transfer instructions from one pipeline stage to the next. Let $Stage_i$ denote the $i$-th stage where $0 \leq i \leq n-1$, and $n_i$ the number of pipeline registers between $Stage_{i-1}$ and $Stage_i$. Let $IR_{i,j}$ denote an instruction register between $Stage_{i-1}$ and $Stage_i$ ($1 \leq i \leq n$, $1 \leq j \leq n_i$). The first stage, i.e., $Stage_0$, fetches an instruction from instruction memory pointed by program counter $PC$, and stores the instruction into the first instruction register $IR_{1,j}$ ($1 \leq j \leq n_1$). Without loss of generality, let us assume that $IR_{i,j}$ has $p$ parent units and $q$ *children* units as shown in Figure 7. During execution the instruction stored in $IR_{i,j}$ is executed at $Stage_i$ and then stored into the next instruction register $IR_{i+1,k}$ ($1 \leq k \leq q$).

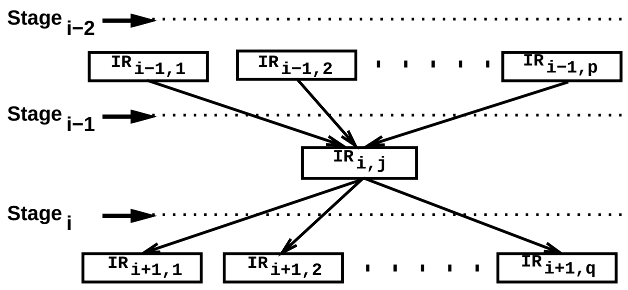

**Figure 7. A fragment of the processor pipeline with only instruction registers**

In this paper, we define a state of the $n$-stage pipeline as values of $PC$ and $\sum_{i=1}^{n-1} n_i$ instruction registers, where $n_i$ is the number of pipeline registers between $Stage_{i-1}$ and $Stage_i$ ($1 \leq i \leq n$). Let $PC(t)$ and $IR_{i,j}(t)$ denote the values of $PC$ and $IR_{i,j}$ at time $t$, respectively. Then, the state of the pipeline at time $t$ is defined as

$$S(t) \;=\; < PC(t), IR_{1,1}(t), \cdots, IR_{i,j}(t), \cdots, IR_{n-1,n_{n-1}}(t) > \tag{1}$$

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

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

**Modeling conditions in FSM**

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

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

For example, if designer specified that "ANY" (see Section 5.1) of the children are responsible for the exception on $IR_{i,j}$ i.e., $IR_{i,j}$ will be in exception condition if any of its children is in exception, the Equation (2) becomes:

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

The condition for squashing ($cond^{SQ}_{IR_{i,j}}$ say) has three components viz., contribution from parents ($NF^{parent}_{IR_{i,j}}$ say), contribution from children ($ST^{child}_{IR_{i,j}}$ say), and self contribution. More formally,

$$cond^{SQ}_{IR_{i,j}} = SQ_{IR_{i,j}} = NF^{parent}_{IR_{i,j}} \wedge ST^{child}_{IR_{i,j}} \wedge ((IR_{i,j}).opcode == nop) \tag{3}$$

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. The $ST_{IR_{i,j}}$ has two components viz., stall condition due to the stall of children ($ST^{child}_{IR_{i,j}}$ say) and stall condition due to hazards or external interrupts 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, $cond^{ST}_{IR_{i,j}}(S(t), I(t))$ ($cond^{ST}_{IR_{i,j}}$ in short),

$$cond^{ST}_{IR_{i,j}} = ST_{IR_{i,j}} = ST^{child}_{IR_{i,j}} \vee ST^{self}_{IR_{i,j}} \tag{4}$$

For example, if designer specified that "ALL" (see Section 5.1) the children are responsible for the stalling of $IR_{i,j}$, i.e., $IR_{i,j}$ is stalled when all of its children are stalled, the Equation (4) becomes:

$$cond^{ST}_{IR_{i,j}} = ST_{IR_{i,j}} = (\bigwedge_{k=1}^{q} ST_{IR_{i+1,k}}) \vee ST^{self}_{IR_{i,j}}$$

However, in the presence of exceptions ($XN_{IR_{i,j}}$) on $IR_{i,j}$ and when bubble squashing ($SQ_{IR_{i,j}}$) is allowed the condition for stalling (Equation (4)) becomes

$$cond^{ST}_{IR_{i,j}} = (ST^{child}_{IR_{i,j}} \vee ST^{self}_{IR_{i,j}}) \wedge \overline{XN_{IR_{i,j}}} \wedge \overline{SQ_{IR_{i,j}}} \tag{5}$$

As mentioned earlier, the $ST^{self}_{IR_{i,j}}$ component of the stall condition is true when there are hazards or exceptions on $IR_{i,j}$. However, for the processor with fragmented pipelines the $ST^{self}_{IR_{i,j}}$ component is also used to describe the instruction dependencies. A fragmented pipeline has fork and join nodes. A node is called a *fork* node in our model if it has more than one children. For example, the node $IR_{i,j}$ in Figure 4 is a fork node since it has $q$ children nodes i.e., it can send instructions to $q$ children nodes. Similarly, a *join* node in the model is a node with more than one parent nodes. For example, the node $IR_{u,v}$ in Figure 4 is a join node since it has $q$ parent nodes i.e., it can receive instructions from $q$ parent nodes. A fork node and its corresponding join node is termed as a *fork-join* pair. For example, $IR_{i,j}$ and $IR_{u,v}$ is a fork-join pair in Figure 4. We compute the $ST^{self}_{IR_{i,j}}$ for fragmented pipelines by examining the instruction flow between fork-join pairs. While issuing an instruction, a fork node $f_{node}$ has to ensure that the instruction will not reach the corresponding join node $j_{node}$ before the instructions issued earlier, the instructions currently in the nodes between $f_{node}$ and $j_{node}$ fork-join pair; otherwise it will be out-of-order execution. Consider, $IR_{i,j}$ (fork node) in Figure 4 that can send instructions to $q$ pipelines and these $q$ pipelines join in $IR_{u,v}$ (join node). When $IR_{i,j}$ decides to send an instruction ($I_k$ say) to the children in $k$-th pipeline, it has to observe the instructions in pipelines $x$ ($x \neq k; 1 \leq (x,k) \leq q$) to ensure in-order execution i.e., the instruction $I_k$ (if issued) should not reach the join node $IR_{u,v}$ before the instructions which are issued earlier. Let us define $\tau(y)$ as the total number of clock cycles needed by pipeline $y$. 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 join node $IR_{u,v}$) and the timing of each pipeline stage. This computation can be performed recursively for pipelines containing fragmented pipelines. Then $ST^{self}_{IR_{i,j}}$ becomes

$$ST^{self}_{IR_{i,j}} = \bigvee_{k=1}^{q} (I_k \wedge (\bigvee_{x=1}^{q} S_{x,k})) \tag{6}$$

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

The condition for normal flow ($cond^{NF}_{IR_{i,j}}$ say) has five components viz., contribution from parents ($NF^{parent}_{IR_{i,j}}$ say), contribution from children ($NF^{child}_{IR_{i,j}}$ say), self contribution (not stalled), not squashed ($SQ_{IR_{i,j}}$), and not in exception ($XN_{IR_{i,j}}$). More formally,

$$cond^{NF}_{IR_{i,j}} = NF^{parent}_{IR_{i,j}} \wedge NF^{child}_{IR_{i,j}} \wedge \overline{ST^{self}_{IR_{i,j}}} \wedge \overline{XN_{IR_{i,j}}} \wedge \overline{SQ_{IR_{i,j}}} \tag{7}$$

For example, if the designer specified that $IR_{i,j}$ will be in normal flow if ”ANY” (see Section 5.1) of its parents is not stalled and ”ANY” of its children is not stalled, the Equation (7) becomes

$$cond^{NF}_{IR_{i,j}} = (\bigvee_{l=1}^{p} \overline{ST_{IR_{i-1,l}}}) \wedge (\bigvee_{k=1}^{q} \overline{ST_{IR_{i+1,k}}}) \wedge \overline{ST^{self}_{IR_{i,j}}} \wedge \overline{XN_{IR_{i,j}}} \wedge \overline{SQ_{IR_{i,j}}}$$

The condition for bubble insertion ($cond^{BI}_{IR_{i,j}}$ say) has five components viz., contribution from parents ($BI^{parent}_{IR_{i,j}}$ say), contribution from children ($BI^{child}_{IR_{i,j}}$ say), self contribution (not stalled), not squashed, and not in exception. More formally,

$$cond^{BI}_{IR_{i,j}} = BI^{parent}_{IR_{i,j}} \wedge BI^{child}_{IR_{i,j}} \wedge \overline{ST^{self}_{IR_{i,j}}} \wedge \overline{XN_{IR_{i,j}}} \wedge \overline{SQ_{IR_{i,j}}} \tag{8}$$

Similarly the conditions for PC viz., $cond^{SE}_{PC}$ (SE: sequential execution), $cond^{BI}_{PC}$ (BI: bubble insertion), and $cond^{BT}_{PC}$ (BT: branch taken) can be described using the information available in the ADL. The $cond^{BT}_{PC}$ is true when a branch is taken or when an exception is taken. When a branch is taken, the PC is modified with the target address. When an exception is taken, the PC is updated with the corresponding interrupt service routine address. Let us assume, $BT_{PC}$ bit is set when the unit completes execution of a branch instruction and the branch is taken. Formally,

$$cond^{SE}_{PC}(S(t), I(t)) = NF^{child}_{PC} \wedge \overline{ST^{self}_{PC}} \wedge \overline{BT_{PC}} \wedge \overline{XN_{IR_{1,j}}} \tag{9}$$

$$cond^{ST}_{PC}(S(t), I(t)) = (ST^{child}_{PC} \vee ST^{self}_{PC}) \wedge \overline{BT_{PC}} \wedge \overline{XN_{IR_{1,j}}} \tag{10}$$

$$cond^{BT}_{PC}(S(t), I(t)) = (BT_{PC} \vee XN_{IR_{1,j}}) \tag{11}$$

## Modeling State Transition Functions

In this section, we describe the next-state function of the FSM. Figure 7 shows a fragment of the processor pipeline with only instruction registers. If there are no pipeline hazards, instructions flow from IR (instruction register) to IR every $m$ cycles (m = 1 for single-cycle IR). In this case, the instruction in $IR_{i-1,l}$ $(1 \leq l \leq p)$ at time $t$ proceeds to $IR_{i,j}$ after $m$ cycles ( $m$ is the *timing* of $IR_{i-1,l}$ and $IR_{i,j}$ has $p$ parent latches and $q$ child latches as shown in Figure 7), i.e., $IR_{i,j}(t+1) = IR_{i-1,l}(t)$. In the presence of pipeline hazards, however, the instruction in $IR_{i,j}$ may be stalled, i.e., $IR_{i,j}(t+1) = IR_{i,j}(t)$. Note that, in general, any instruction in the pipeline cannot skip 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.

The rest of this section formally describes the next-state function of the FSM. According to the Equation (1), a state of an $n-$stage pipeline is defined by $(M+1)$ registers (PC and $M$ instruction registers where, $M=\sum_{i=1}^{n-1} n_i$). Therefore, the next state function of the pipeline can also be decomposed into $(M+1)$ sub-functions each of which is dedicated to a specific state register. Let $f_{PC}^{NS}$ and $f_{IR_{i,j}}^{NS}$ ($1 \leq i \leq n-1$, $1 \leq j \leq n_i$) 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.

$$
\begin{aligned}
&PC(t+1)\\
&\quad = f_{PC}^{NS}(S(t),I(t))\\
&\quad = \begin{cases} PC(t)+L & \text{if } cond_{PC}^{SE}(S(t),I(t))=1\\ target & \text{if } cond_{PC}^{BT}(S(t),I(t))=1\\ PC(t) & \text{if } cond_{PC}^{ST}(S(t),I(t))=1 \end{cases}
\end{aligned} \tag{12}
$$

Here, $I(t)$ represents a set of external signals at time $t$, $L$ represents the instruction length, and *target* represents the branch target address which is computed at a certain pipeline stage. The $cond_{PC}^{x}$'s ($x \in SE, BT, ST$) are logic functions of $S(t)$ and $I(t)$ as described in Equation (9) - Equation (11), and return either 0 or 1. For example, if $cond_{PC}^{ST}(S(t),I(t))$ is 1, $PC$ keeps its current value at the next cycle.

For the first instruction register, $IR_{1,j}$ ($1 \leq j \leq n_1$), we define the following five types of state transitions. The *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 $cond_{IR_{1,1}}^{NF}(S(t),I(t))$ is 1, an instruction is fetched from instruction memory and stored into $IR_{1,1}$. If $cond_{IR_{1,1}}^{ST}(S(t),I(t))$ is 1, $IR_{1,1}$ remains unchanged.

$$
\begin{aligned}
&IR_{1,j}(t+1)\\
&\quad = f_{IR_{1,j}}^{NS}(S(t),I(t))\\
&\quad = \begin{cases} IM(PC(t)) & \text{if } cond_{IR_{1,j}}^{NF}(S(t),I(t))=1\\ IR_{1,j}(t) & \text{if } cond_{IR_{1,j}}^{ST}(S(t),I(t))=1\\ nop & \text{if } cond_{IR_{1,j}}^{BI}(S(t),I(t))=1\\ IM(PC(t)) & \text{if } cond_{IR_{1,j}}^{SQ}(S(t),I(t))=1\\ nop & \text{if } cond_{IR_{1,j}}^{XN}(S(t),I(t))=1 \end{cases}
\end{aligned} \tag{13}
$$

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

$$
\begin{aligned}
&IR_{i,j}(t+1)\\
&\quad = f_{i,j}^{NS}(S(t),I(t))
\end{aligned}
$$

$$= \begin{cases} IR_{i-1,l}(t) & \text{if } cond^{NF}_{IR_{i,j}}(S(t),I(t)) = 1 \\ IR_{i,j}(t) & \text{if } cond^{ST}_{IR_{i,j}}(S(t),I(t)) = 1 \\ nop & \text{if } cond^{BI}_{IR_{i,j}}(S(t),I(t)) = 1 \\ IR_{i-1,l}(t) & \text{if } cond^{SQ}_{IR_{i,j}}(S(t),I(t)) = 1 \\ nop & \text{if } cond^{XN}_{IR_{i,j}}(S(t),I(t)) = 1 \end{cases} \tag{14}$$

The $IR_{i,j}$ 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 (bubble) is inserted in $IR_{i,j}$ when $cond^{BI}_{IR_{i,j}}(S(t),I(t))$ or $cond^{XN}_{IR_{i,j}}(S(t),I(t))$ is 1, resulting in $IR_{i,j}(t+1) = nop$. Similarly, when $cond^{SQ}_{IR_{i,j}}(S(t),I(t))$ is 1, the bubble in $IR_{i,j}$ gets overwritten by the instruction from the parent instruction register, i.e., $IR_{i,j}(t+1) = IR_{i-1,l}(t)$ $(1 \le l \le n_{i-1})$.

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

## 6 Validation of Processor Specification

Based on the FSM modeling presented in Section 5, we propose a method for validating pipelined processor specifications using three properties: determinism, in-order execution and finiteness. We first describe the properties needed for validating the specification , then we present an automatic property checking framework driven by EXPRESSION ADL [1].

### 6.1 Properties

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

**Determinism**

To ensure correct execution, there should not be any instruction or data loss in the pipeline. The bubble squashing and flushing of instructions are permitted. The flushed instructions are fetched and executed again. The next-state functions for all state registers must be deterministic. This property is valid if all the following equations hold.

$$cond^{SE}_{PC} \vee cond^{BT}_{PC} \vee cond^{ST}_{PC} = 1 \tag{15}$$

$$\forall i,j(1 \le i \le n-1, 1 \le j \le n_i),$$
$$cond^{NF}_{IR_{i,j}} \vee cond^{ST}_{IR_{i,j}} \vee cond^{BI}_{IR_{i,j}} \vee cond^{XN}_{IR_{i,j}} \vee cond^{SQ}_{IR_{i,j}} = 1 \tag{16}$$

$$\forall x,y(x,y \in \{SE,BT,ST\} \ \wedge \ x \ne y),\ cond^{x}_{PC} \wedge cond^{y}_{PC} = 0 \tag{17}$$

$$\forall i,j(1 \le i \le n-1, 1 \le j \le n_i),$$
$$\forall x,y(x,y \in \{NF,ST,BI,XN,SQ\} \ \wedge \ x \ne y),\ cond^{x}_{IR_{i,j}} \wedge cond^{y}_{IR_{i,j}} = 0 \tag{18}$$

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

**In-Order Execution**

A pipelined processor with in-order execution is correct if all instructions which are fetched from instruction memory flow from the first stage to the last stage while maintaining their execution order. In order to guarantee in-order execution, state transitions of adjacent instruction registers must depend on each other. Illegal combination of state transitions of adjacent stages are described below using Figure 7.

- An instruction register can not be in normal flow if all the parent instruction registers (adjacent ones) are stalled. If such a combination of state transitions are allowed, the instruction stored in $IR_{i-1,l}$ ($1 \leq l \leq p$) at time $t$ will be duplicated, and stored into both $IR_{i-1,l}$ and $IR_{i,j}$ in the next cycle. Therefore, the instruction will be executed more than once. Formally, the Equation (19) should be satisfied.

$$\left(\bigwedge_{l=1}^{p} cond^{ST}_{IR_{i-1,l}}\right) \wedge cond^{NF}_{IR_{i,j}} = 0 \qquad (19)$$
$$(2 \leq i \leq n-1, 1 \leq j \leq n_i, 1 \leq l \leq p)$$

- 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. Formally, the Equation (20) should be satisfied.

$$cond^{NF}_{IR_{i,j}} \wedge \left(\bigwedge_{k=1}^{q} cond^{ST}_{IR_{i+1,k}}\right) = 0 \qquad (20)$$
$$(2 \leq i \leq n-1, 1 \leq j \leq n_i, 1 \leq k \leq q)$$

- Similarly, if $IR_{i,j}$ is in bubble insertion, at least one of its child latches should not be stalled. If all of its child latches are stalled, the instruction stored in $IR_{i+1,k}$ ($1 \leq k \leq q$) at time $t$ will be overwritten by the bubble. Formally, the Equation (21) should be satisfied.

$$cond^{BI}_{IR_{i,j}} \wedge \left(\bigwedge_{k=1}^{q} cond^{ST}_{IR_{i+1,k}}\right) = 0 \qquad (21)$$
$$(2 \leq i \leq n-1, 1 \leq j \leq n_i, 1 \leq k \leq q)$$

- Similarly, if $IR_{i,j}$ is in bubble insertion condition, Equation (22) - Equation (23) should be satisfied.

$$cond^{NF}_{IR_{i-1,l}} \wedge cond^{BI}_{IR_{i,j}} = 0 \qquad (22)$$
$$cond^{BI}_{IR_{i-1,l}} \wedge cond^{BI}_{IR_{i,j}} = 0 \qquad (23)$$
$$(2 \leq i \leq n-1, 1 \leq j \leq n_i, 1 \leq l \leq p)$$

- Similarly, if $IR_{i,j}$ is in bubble squash state, Equation (24) - Equation (27) describes the illegal combination of state transitions of adjacent instruction registers.

$$cond^{ST}_{IR_{i-1,l}} \wedge cond^{SQ}_{IR_{i,j}} = 0 \tag{24}$$
$$cond^{XN}_{IR_{i-1,l}} \wedge cond^{SQ}_{IR_{i,j}} = 0 \tag{25}$$
$$cond^{SQ}_{IR_{i,j}} \wedge cond^{NF}_{IR_{i+1,k}} = 0 \tag{26}$$
$$cond^{SQ}_{IR_{i,j}} \wedge cond^{NI}_{IR_{i+1,k}} = 0 \tag{27}$$
$$(2 \le i \le n-1, 1 \le j \le n_i, 1 \le l \le p, 1 \le k \le q)$$

- Finally, an instruction register can not be in normal flow, stall, squash, or bubble insertion if next (child) instruction register is in exception. Formally, the Equation (28) - Equation (31) should be satisfied.

$$cond^{NF}_{IR_{i-1,l}} \wedge cond^{XN}_{IR_{i,j}} = 0 \tag{28}$$
$$cond^{ST}_{IR_{i-1,l}} \wedge cond^{XN}_{IR_{i,j}} = 0 \tag{29}$$
$$cond^{SQ}_{IR_{i-1,l}} \wedge cond^{XN}_{IR_{i,j}} = 0 \tag{30}$$
$$cond^{BI}_{IR_{i-1,l}} \wedge cond^{XN}_{IR_{i,j}} = 0 \tag{31}$$
$$(2 \le i \le n-1, 1 \le j \le n_i, 1 \le l \le p)$$

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 $I_b$. Formally the following equation should hold:

$$\forall (F,J), I_a \preceq_J I_b \Rightarrow \Gamma_F(I_a) < \Gamma_F(I_b) \tag{32}$$

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

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

$$\forall x,y(x,y \in \{1,2,...,p\} \ \wedge \ x \neq y), \ cond^{NF}_{IR_{i-1,x}} \wedge cond^{NF}_{IR_{i-1,y}} = 0 \tag{33}$$

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 between $PC$ and $IR_{1,j}$ are described below.

$$cond^{ST}_{PC} \wedge cond^{NF}_{IR_{1,j}} = 0 \tag{34}$$
$$cond^{SE}_{PC} \wedge (\bigwedge_{j=1}^{n_1} cond^{ST}_{IR_{1,j}}) = 0 \tag{35}$$
$$cond^{BT}_{PC} \wedge (\bigwedge_{j=1}^{n_1} cond^{ST}_{IR_{1,j}}) = 0 \tag{36}$$

$$cond_{PC}^{SE} \wedge cond_{IR_{1,j}}^{BI} = 0 \tag{37}$$
$$cond_{PC}^{BT} \wedge cond_{IR_{1,j}}^{BI} = 0 \tag{38}$$
$$cond_{PC}^{SE} \wedge cond_{IR_{1,j}}^{XN} = 0 \tag{39}$$
$$cond_{PC}^{ST} \wedge cond_{IR_{1,j}}^{SQ} = 0 \tag{40}$$
$$cond_{PC}^{ST} \wedge cond_{IR_{1,j}}^{XN} = 0 \tag{41}$$

We have described all possible illegal combination of state transition functions (Equation (19) - Equation (41)). However, Equation (22), Equation (23), Equation (26), and Equation (27) are not necessary to prove 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 may be stalled indefinitely. Therefore, we need to guarantee that stall conditions (i.e., $cond_{IR_{i,j}}^{ST}$) are resolved in a finite number of cycles. The Equation (4) (stall condition) has 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.
- $cond_{IR_{i,j}}^{ST}$ can be a function of external signals and $IR_{k,y}$ where $k \geq i$, but cannot be a function of $IR_k$ where $k < i$.

## 6.2 Automatic Validation Framework

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

```
Algorithm 1: Validation of Pipeline Specification
Input: ADL specification of the processor architecture.
Outputs: Success, if the processor model satisfies the properties (determinism, in-order execution, finiteness).
         Failure otherwise, and produces the counter-examples.
Begin
    Generate FSM model from the ADL specification using Equation (1) - Equation (14)
    Generate properties using Equation (15) - Equation (41)
    Apply the properties on the FSM model to verify determinism, in-order execution, and finiteness.
    Return Success if all the properties are verified;
           Failure otherwise, and produce the counter-example(s).
End
```

We have verified the properties using two different approaches. First, we have used SMV [38] based property checking framework as shown in Figure 8. The SMV based approach fits nicely in our validation framework. However, the SMV is limited by the size of design it can handle. We have also developed an equation solver based framework as shown in Figure 9 that can handle complex designs. In this section, we briefly describe these two approaches used in our framework. The detailed description is available in [28].

### Validation using Model Checker

The FSM model (SMV description) of the processor is generated from the ADL specification. The properties are also described using SMV description. The properties are applied on the FSM model using the 1SMV model checker as shown in Figure 8. In case of failure, SMV generates counter examples that can be used to modify the ADL specification. Each counter example describes the failed equation(s) and the instructions registers that are involved.

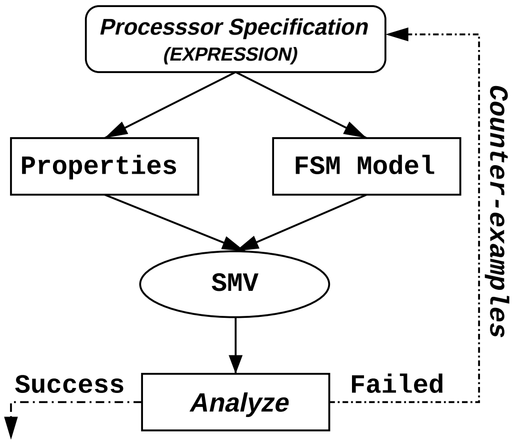

**Figure 8. Validation Framework using SMV**

We have verified the in-order execution style of the processor specification in two ways. First, the framework generates properties using Equation (19) - Equation (41) to verify in-order execution. This is similar to how other properties (determinism and finiteness) are verified. Second, an auxiliary automata is used instead of using equations to verify in-order execution. In auxiliary automata based approach, we use the same FSM model of the processor (SMV description) generated from the ADL. We developed a SMV module that generates two instructions randomly with random delay between them. These two instructions are recorded and fed to the FSM model. The processor (FSM) model accepts these instructions and performs regular computations. At the completion (e.g., writeback unit) the auxiliary automata analyzes these two instructions to see whether they completed in the same sequence as generated. Note that, this auxiliary automata does not need any manual modification for different architectures. In case of failure SMV generates counter-examples containing instruction sequence (instruction pair with NOPs in between them) that violates in-order execution for the processor model.

### Validation using Equation Solver

In the second approach, the framework generates the FSM model and flow equations for NF, ST, XN, SQ, and BI for each instruction register and SE, ST, and BT for PC using ADL description and Equation (1) - Equation (14). It generates the equations necessary for verifying properties using ADL description and Equation (15) - Equation (41) as shown in Figure 9.

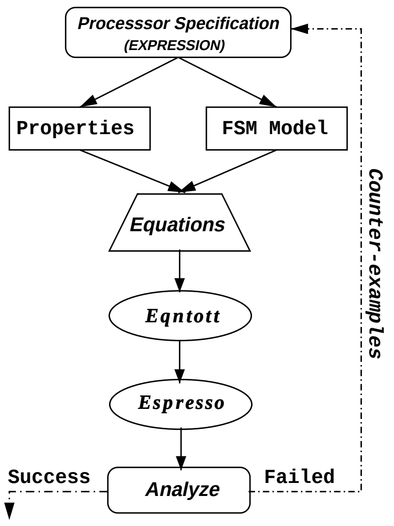

**Figure 9. Validation Framework using Equation Solver**

The *Eqntott* [11] tool converts these equations in two-level representation of a two-valued Boolean function. This two-level representation is fed to *Espresso* [10] tool that produces minimal equivalent representation. Finally, the minimized representation is analyzed to determine whether the property is successfully verified or not. In case of failure it generates traces explaining the cause of failure. The trace contains the equation(s) that failed, and the identification of the instruction registers involved. The designer knows the property that is violated and the reason of the violation. This information is used to modify the ADL specification. The detailed description is available in [28].

## 7 A Case Study

In a case study we successfully applied the proposed methodology to the single-issue DLX [17] processor. We have chosen DLX processor since it has been well studied in academia and contains many interesting features viz., fragmented pipelines, multicycle units etc., that are representative of many commercial pipelined processor architectures such as TI C6x [41], PowerPC [12], and MIPS R10K [13]. Figure 10 shows the DLX processor pipeline. The DLX architecture has five pipeline stages: fetch (IF), decode (ID), execute, memory (MEM), and writeback (WB). The *execute* stage has four parallel pipeline paths: integer ALU (EX), 7 stage multiplier (M1 - M7), four stage floating-point adder (A1 - A4), and multi-cycle divider (DIV).

We used the EXPRESSION ADL [1] to capture the structure and behavior of the DLX processor. We captured the conditions for stalling, normal flow, exception, branch taken, squashing, and bubble insertion in the ADL. 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 captured other flow conditions. For example, the condition specification for the decode unit is shown below. The definitions of (*ALL*, *ANY*, *SOME*, *NONE*) are described in Section 5.1.

```
(DecodeUnit DECODE
    .........
    (CONDITIONS
        (NF ANY ANY)
```

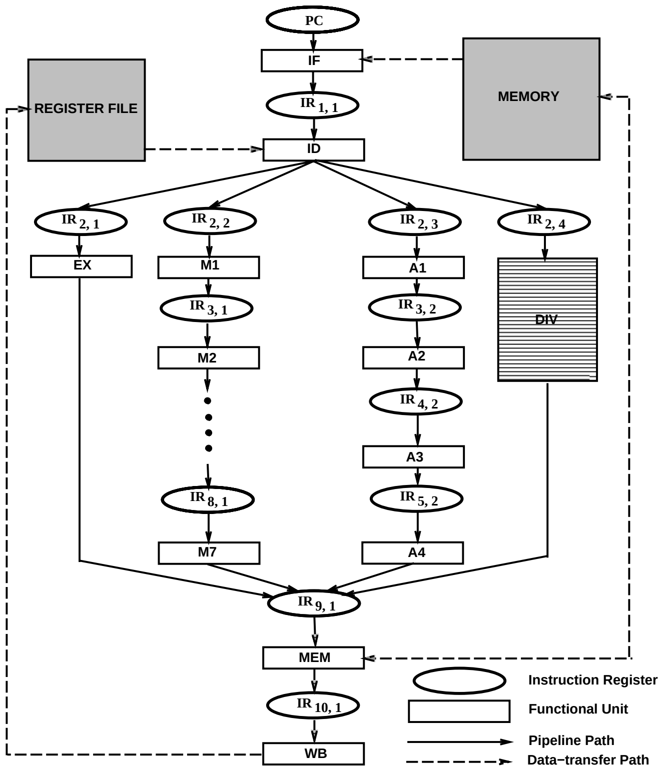

**Figure 10. The DLX Processor**

```
        (ST ALL)
        (XN ANY)
        (BI ALL ANY)
        (SQ ANY ALL)
        (SELF "")
    )
)
```

Using the ADL description, we automatically generated the equations for flow conditions for all the units [28]. For example, the flow equations for the decode latch are shown below (using Equation (2) - Equation (8), and the description of the decode unit shown above).

$$cond^{NF}_{IR_{1,1}} = \overline{ST_{PC}} \wedge (\overline{ST_{IR_{2,1}}} \vee \overline{ST_{IR_{2,2}}} \vee \overline{ST_{IR_{2,3}}} \vee \overline{ST_{IR_{2,4}}}) \wedge \overline{XN_{IR_{1,1}}} \wedge \overline{SQ_{IR_{1,1}}} \tag{42}$$

$$cond^{ST}_{IR_{1,1}} = (ST_{IR_{2,1}} \wedge ST_{IR_{2,2}} \wedge ST_{IR_{2,3}} \wedge ST_{IR_{2,4}}) \wedge \overline{XN_{IR_{1,1}}} \wedge \overline{SQ_{IR_{1,1}}} \tag{43}$$

$$cond^{NI}_{IR_{1,1}} = ST_{PC} \wedge (\overline{ST_{IR_{2,1}}} \vee \overline{ST_{IR_{2,2}}} \vee \overline{ST_{IR_{2,3}}} \vee \overline{ST_{IR_{2,4}}}) \wedge \overline{XN_{IR_{1,1}}} \wedge \overline{SQ_{IR_{1,1}}} \tag{44}$$

$$cond^{XN}_{IR_{1,1}} = XN_{IR_{1,1}} = XN_{IR_{2,1}} \vee XN_{IR_{2,2}} \vee XN_{IR_{2,3}} \vee XN_{IR_{2,4}} \tag{45}$$

$$cond^{SQ}_{IR_{1,1}} = SQ_{IR_{1,1}} = \overline{ST_{PC}} \wedge (ST_{IR_{2,1}} \wedge ST_{IR_{2,2}} \wedge ST_{IR_{2,3}} \wedge ST_{IR_{2,4}}) \wedge ((IR_{1,1}).opcode == nop) \tag{46}$$

The $IR_{2,4}$ represents the latch for the multicycle unit. So we assumed a signal *busy*, internal to $IR_{2.4}$, which remained set for *m* cycles. The *busy* can be treated as $ST^{self}_{IR_{2.4}}$ as shown in Equation (5).

The necessary equations for verifying the properties viz., determinism, in-order execution and finiteness are generated automatically from the given ADL specification. 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:

$cond_{IR_{1,1}}^{NF} \vee cond_{IR_{1,1}}^{ST} \vee cond_{IR_{1,1}}^{BI} \vee cond_{IR_{1,1}}^{XN} \vee cond_{IR_{1,1}}^{SQ}$
$= (\overline{ST_{PC}} \wedge (\overline{ST_{IR_{2,1}}} \vee \overline{ST_{IR_{2,2}}} \vee \overline{ST_{IR_{2,3}}} \vee \overline{ST_{IR_{2,4}}}) \wedge \overline{XN_{IR_{1,1}}} \wedge \overline{SQ_{IR_{1,1}}}) \vee ((ST_{IR_{2,1}} \wedge ST_{IR_{2,2}} \wedge ST_{IR_{2,3}} \wedge ST_{IR_{2,4}}) \wedge$
$\overline{XN_{IR_{1,1}}} \wedge \overline{SQ_{IR_{1,1}}}) \vee (ST_{PC} \wedge (\overline{ST_{IR_{2,1}}} \vee \overline{ST_{IR_{2,2}}} \vee \overline{ST_{IR_{2,3}}} \vee \overline{ST_{IR_{2,4}}}) \wedge \overline{XN_{IR_{1,1}}} \wedge \overline{SQ_{IR_{1,1}}}) \vee XN_{IR_{1,1}} \vee SQ_{IR_{1,1}}$
$= ((\overline{ST_{IR_{2,1}}} \vee \overline{ST_{IR_{2,2}}} \vee \overline{ST_{IR_{2,3}}} \vee \overline{ST_{IR_{2,4}}}) \wedge (\overline{ST_{PC}} \vee ST_{PC}) \wedge \overline{XN_{IR_{1,1}}} \wedge \overline{SQ_{IR_{1,1}}}) \vee ((ST_{IR_{2,1}} \wedge ST_{IR_{2,2}} \wedge ST_{IR_{2,3}} \wedge$
$ST_{IR_{2,4}}) \wedge \overline{XN_{IR_{1,1}}} \wedge \overline{SQ_{IR_{1,1}}}) \vee XN_{IR_{1,1}} \vee SQ_{IR_{1,1}}$
$= \overline{XN_{IR_{1,1}}} \wedge \overline{SQ_{IR_{1,1}}} \wedge ((\overline{ST_{IR_{2,1}}} \vee \overline{ST_{IR_{2,2}}} \vee \overline{ST_{IR_{2,3}}} \vee \overline{ST_{IR_{2,4}}}) \vee (ST_{IR_{2,1}} \wedge ST_{IR_{2,2}} \wedge ST_{IR_{2,3}} \wedge ST_{IR_{2,4}})) \vee XN_{IR_{1,1}} \vee$
$SQ_{IR_{1,1}}$
$= (\overline{XN_{IR_{1,1}}} \wedge \overline{SQ_{IR_{1,1}}}) \vee XN_{IR_{1,1}} \vee SQ_{IR_{1,1}}$
$= 1$

We have used *Espresso* to minimize the equations. These minimized equations are analyzed to verify whether the properties are violated or not. Our framework determined that the Equation (32) is violated and generated a simple instruction sequence which violates in-order execution: floating-point addition followed by integer addition. The decode unit issued floating point addition $I_{fadd}$ operation in cycle $m$ to floating-point adder pipeline (A1 - A4) and an integer addition operation $I_{iadd}$ to integer ALU (EX) at cycle *m+1*. The instruction $I_{iadd}$ reached join node (MEM unit) prior to $I_{fadd}$.

We modified the ADL description to change the stall condition depending on current instruction in decode unit and the instructions active in the integer ALU, MUL, FADD, and DIV pipelines using Equation (6). The current instruction will not be issued (decode stalls) if it leads to out-of-order execution. Our framework generated equations for the modified processor model. The only difference is $ST_{IR_{i,j}}^{self}$ for decode unit (Equation (6)) becomes: $ST_{IR_{i,j}}^{self}$ =

$$(I_1 \wedge (S_{2,1} \vee S_{3,1} \vee S_{4,1})) \vee I_2(\wedge S_{4,2}) \vee I_3(\wedge(S_{2,3} \vee S_{4,3}))$$

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 an instruction in DIV unit which is active for less than 18 cycles ($\tau(4) - \tau(2)$ = 25 - 7 = 18), decode should stall to avoid 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 do not have four $S_{x,y}$ symbols.

The Equation (33) 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 $m$) followed by a floating-point add operation (issued by decode unit in cycle *(m + 3)*). As a result both the operations reached $IR_{9,1}$ at cycle *(m+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.

The framework determined that the in-order execution is violated in the presence of an exception in

A2 stage of the floating-point adder and generated a simple instruction sequence which violates in-order execution: floating-point addition followed by integer addition. The decode unit issued floating point addition $I_{fadd}$ operation in cycle *m* to floating-point adder pipeline (A1 - A4) and an integer addition operation $I_{iadd}$ to integer ALU (EX) at cycle *m+1*. Due to an exception in A2 stage, the A1, ID and IF stages got flushed whereas the $I_{iadd}$ operation continued its execution.

We modified the ADL description to incorporate the notion of in-order relationship among units. For example, for *EX* unit the in-order children are *M2* and *A2*. Note that, when M3 stage is in exception condition, the M2 stage and its parent (normal parent M1 and in-order parents EX and A1) stages will be flushed. Similarly, the in-order children for M2 stage is A2 and so on. However, this modeling is not good enough in the presence of multicycle functional units (e.g., DIV unit). The in-order children (parent) information will change depending on how many cycles the division operation is in DIV unit. So we model division unit with m-stages ($D_1$, $D_2$, .., $D_n$), where *m* is the latency of the division operation, with the assumption that the stage $D_i$ will have the division operation after *i* cycles and only one stage will have a valid operation at a time. Now we can extend the in-order child concept for multicycle units as well. Now, the in-order children for EX unit are M2, A2 and D2. Similarly, the in-order children for D1 unit are A2 and M2. Since we are considering single-issue machine, two units in the same level will not have valid instruction at the same point in time. For example, M1 and A1 cannot have valid instructions at the same point in time. The in-order execution was successful for this modeling in the presence of exceptions. This modeling is sufficient to handle multiple exceptions where the exception closer to completion has the higher priority. This is due to the fact that the exception closer to completion will flush all the operations above and thereby masks all the exceptions generated after it. For example, if there are exceptions in M3 and A1. The exception in M3 will mask the exception in A1 stage.

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 exploration iterations and varieties increases, the potential for introducing bugs also increases.

We have verified the properties using two different methods: using *SMV* model checker and *Espresso* equation solver, as described in Section 6.2. We have used 296 MHz Sun UltraSparc-II with 1024M RAM to run the experiments. Table 1 shows the performance of the two methods for verifying in-order execution property. We have used the DLX architecture (Figure 10) as the base configuration and modified the number of opcodes. The first column presents our two methods of specification validation. The second, third, and fourth column presents the execution time (in seconds) of the two methods for verifying in-order execution property for different architecture configurations.

We have performed experiments by modifying the pipeline structure: adding pipeline paths, adding pipeline stages etc. However, our SMV based framework could not handle configurations when pipeline path is added due to space explosion in SMV. Our equation solver based framework can handle complex configurations. The SMV based framework took 0.8 seconds to verify determinism property, whereas the equation solver based framework took 4 seconds for the base DLX configuration.

| Methods | DLX Processor Configurations | | |
|---|---|---|---|
| | 8 opcodes | 16 opcodes | 32 opcodes |
| *SMV based Framework* | 302.4 sec | 400.4 sec | 740.9 sec |
| *Espresso based Framework* | 5.4 sec | 6.7 sec | 9.4 sec |

**Table 1. Validation of In-Order Execution by Two Frameworks**

We have used our validation technique in the EXPRESSION [1] framework. The EXPRESSION framework allows verification, automatic software toolkit generation, and design space exploration for a wide range (DSP, VLIW, EPIC, Superscalar) of programmable embedded systems. The EXPRESSION ADL has been used to generate compiler, simulator, and assembler for the DLX [17], TI C6x [41], PowerPC [12], ARM [42], Sun UltraSparc-III [14], and MIPS R10K [13] architectures. The correct execution of the generated compiler and simulator is another indication of the validity of our verification approach.

## 8 Summary

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

We have developed validation techniques to ensure that the static behavior of the pipeline is well-formed by analyzing the structural aspects of the specification using a graph based model [32]. In this paper, we verify the dynamic behavior by analyzing the instruction flow in the pipeline using a Finite State Machine (FSM) based model to validate several important architectural properties such as determinism, finiteness, and execution style (e.g., in-order execution) in the presence of hazards and multiple exceptions. These properties are by no means complete to prove the correctness of the specification. The designer can add new architecture specific properties and easily integrate it in our framework. Our validation framework uses two approaches: SMV based property checking and Espresso based equation minimization. The framework determines if all the necessary properties are satisfied or not. In case of a failure, it generates traces so that designer can modify the ADL specification of the architecture. We applied this methodology to the DLX processor to demonstrate the usefulness of our approach.

Currently, we can model and verify single-issue microprocessors with in-order execution, fragmented pipelines and multicycle functional units [29] in the presence of hazards and multiple exceptions [30]. Our future work will extend modeling and validation technique towards superscalar processors.

## 9 Acknowledgments

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

## References

[1] A. Halambi and P. Grun and V. Ganesh and A. Khare and N. Dutt and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. DATE*, Mar. 1999.

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

[3] Axys Design Automation. *http://www.axysdesign.com*.

[4] C. Jacobi. Formal Verification of Complex Out-of-Order Pipelines by Combining Model-Checking and Theorem-Proving. In *CAV*, 2002.

[5] C. Siska. A Processor Description Language Supporting Retargetable Multi-pipeline DSP Program Development Tools. In *Proc. ISSS*, Dec. 1998.

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

[7] G. Goosens et al. CHESS: Retargetable Code Generation for Embedded DSP Processors. In *Code Generation for Embedded Processors.* Kluwer, 1997.

[8] G. Hadjiyiannis et al. ISDL: An Instruction Set Description Language for Retargetability. In *DAC*, 1997.

[9] H. Yasuura et al. A Programming Language for Processor Based Embedded Systems. In *Proc. APCHDL*, 1998.

[10] http://www-cad.eecs.berkeley.edu/Software/software.html. *Espresso*.

[11] http://www-ee.engr.ccny.cuny.edu/notes/ee210/eqntott_man.html. *Eqntott*.

[12] http://www.motorola.com/SPS/PowerPC. *MPC7400 PowerPC Microprocessor*.

[13] http://www.sgi.com/processors/r10k. *MIPS R10000 Microprocessor*.

[14] http://www.sun.com/microelectronics/UltraSparc-III. *UltraSparc III*.

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

[16] J. Hauke and J. Hayes. Specification and verification of pipelining in the ARM2 RISC microprocessor. In *ACM TODAES*, volume 3(4), pages 563–580, October 1998.

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

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

[19] J. Sawada and W. D. Hunt. Processor Verification with Precise Exceptions and Speculative Execution. In *CAV*, 1998.

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

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

[22] M. Aagaard et al. *A Framework for Microprocessor Correctness Statements*. CHARME, 2001.

[23] M. Freericks. The nML Machine Description Formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Dept., 1993.

[24] M. N. Velev. Formal Verification of VLIW Microprocessors with Speculative Execution. In *CAV*, 2000.

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

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

[27] P. Ho et al. Formal Verification of Pipeline Control Using Controlled Token Nets and Abstract Interpretation. In *ICCAD*, 1998.

[28] P. Mishra and H. Tomiyama and N. Dutt and A. Nicolau. Architecture Description Language driven Verification of In-Order Execution in Pipelined Processors. Technical Report UCI-ICS 01-20, University of California, Irvine, 2000.

[29] P. Mishra and H. Tomiyama and N. Dutt and A. Nicolau. Automatic Verification of In-Order Execution in Microprocessors with Fragmented Pipelines and Multicycle Functional Units. In *DATE*, 2002.

[30] P. Mishra and N. Dutt. Modeling and Verification of Pipelined Embedded Processors in the Presence of Hazards and Exceptions. In *IFIP WCC DIPES*, 2002.

[31] P. Mishra et al. Specification of Hazards, Stalls, Interrupts, and Exceptions in EXPRESSION. Technical Report UCI-ICS 01-05, University of California, Irvine, 2001.

[32] P. Mishra et al. Automatic Modeling and Validation of Pipeline Specifications driven by an Architecture Description Language. In *ASPDAC / VLSI Design*, 2002.

[33] P. Paulin et al. FlexWare: A Flexible Firmware Development Environment for Embedded Systems. In *Proc. Dagstuhl Code Generation Workshop*, 1994.

[34] R. Jhala and K. L. McMillan. Microarchitecture Verification by Compositional Model Checking. In *CAV*, 2001.

[35] R. Leupers and P. Marwedel. Retargetable Code Generation based on Structural Processor Descriptions. *Design Automation for Embedded Systems*, 3(1), 1998.

[36] R. Leupers et al. Retargetable Generation of Code Selectors from HDL Processor Models. In *Proc. EDTC*, 1997.

[37] R. M. Hosabettu. *Systematic Verification Of Pipelined Microprocessors*. PhD thesis, PhD Thesis, Department of Computer Science, University of Utah, 2000.

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

[39] Target Compiler Technologies. *http://www.retarget.com*.

[40] Tensilica Inc. *http://www.tensilica.com*.

[41] Texas Instruments. *TMS320C6201 CPU and Instruction Set Reference Guide*, 1998.

[42] The ARM7 User Manual. *http://www.arm.com*.

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

[44] V. Rajesh and Rajat Moona. Processor Modeling for Hardware Software Codesign. In *International Conference on VLSI Design*, Jan. 1999.

[45] 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.