Modeling and Validation of Pipeline Specifications

PRABHAT MISHRA and NIKIL DUTT
University of California, Irvine

Verification is one of the most complex and expensive tasks in the current Systems-on-Chip design process. Many existing approaches employ a bottom-up approach to pipeline validation, 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 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 pipeline validation. The most important requirement in top-down validation process is to ensure that the specification (reference model) is golden. This paper addresses automatic validation of processor, memory, and coprocessor pipelines described in an ADL. We present a graph-based modeling that captures both structure and behavior of the architecture. Based on this model, we present algorithms to ensure that the static behavior of the pipeline is well formed by analyzing the structural aspects of the specification. We applied our methodology to verify specification of several realistic architectures from different architectural domains to demonstrate the usefulness of our approach.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General—*Modeling of computer architecture*; C.0 [**Computer Systems Organization**]: General—*Systems specification methodology*; I.6.4 [**Computing Methodologies**]: Simulation and Modeling—*Model validation and analysis*

General Terms: Verification, Algorithms, Design, Languages

Additional Key Words and Phrases: Modeling of processor pipeline, pipelined processor specification, pipeline validation, architecture description language

1. INTRODUCTION

Traditional embedded systems consist of programmable processors, coprocessors, application specific integrated circuits (ASIC), memories, and input/output interfaces. Figure 1 shows a traditional hardware/software codesign flow. The embedded system is specified in a system design language. The specification is then partitioned into tasks that are either assigned to software or hardware (ASIC) based on design constraints (cost, power, and performance). Tasks assigned to hardware are translated into hardware description language (HDL) descriptions and then synthesized into ASICs. The tasks assigned to software

Authors' address: Center for Embedded Computer Systems, 444 Computer Science, University of California, Irvine, CA 92697-3430; email: pmishra@ics.uci.edu.

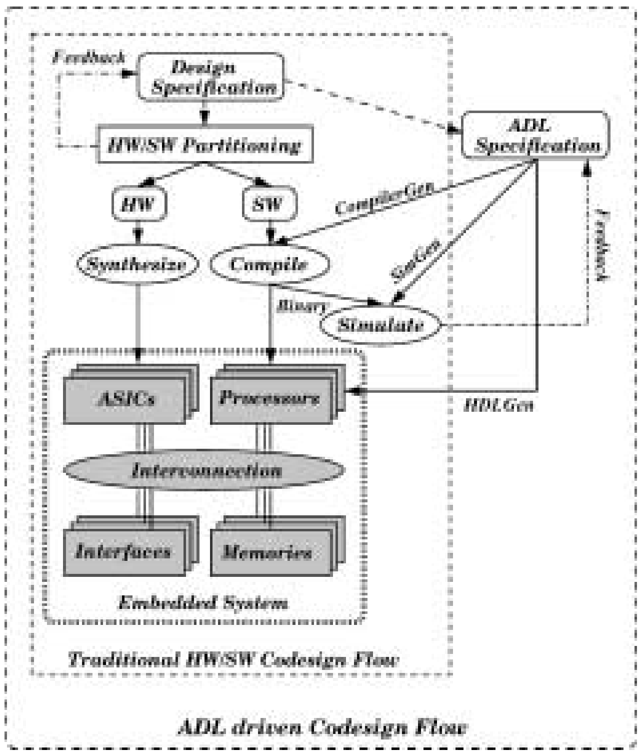

Fig. 1. Hardware/software codesign flow for embedded systems.

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.

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, and 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 of the embedded system is specified in an ADL and the software toolkit can be generated from this description. Figure 2 shows a simplified framework for ADL driven exploration. During early DSE, the system designer modifies the ADL specification of the architecture to exploit the application (software tasks) behavior. The ADL driven DSE has been addressed extensively in both academia: ISDL

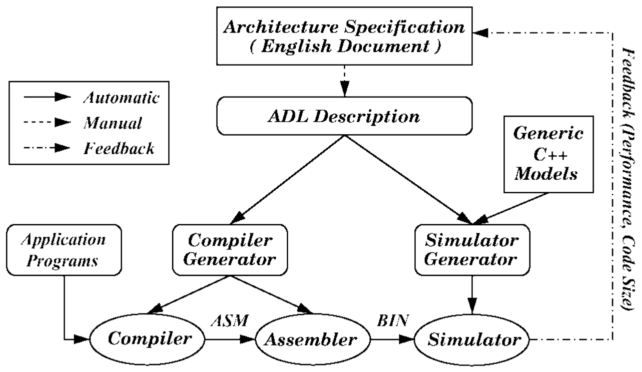

Fig. 2. ADL driven design space exploration.

[Hadjiyiannis et al. 1997], Valen-C [Inoue et al. 1998], MIMOLA [Leupers and Marwedel 1998], LISA [Zivojnovic et al. 1996], nML [Freericks 1993; Rajesh and Moona 1999], and industry: ARC [ARC], Axys [Axys], RADL [Siska 1998], Target [Target], Tensilica [Tensilica], MDES [Trimaran 1997]. The ADL specification is used to generate software toolkit including compiler and simulator to enable architectural exploration. The ADL specification has also been used to generate hardware implementations for rapid system prototyping [Leupers and Marwedel 1998; Mishra et al. 2003; Schliebusch et al. 2002].

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 and hardware implementation. The benefits of validation are twofold. First, the process of any specification is error prone and thus verification techniques can be used to check for correctness and consistency of specification. Second, changes made to the processor during exploration may result in incorrect execution of the system and verification techniques can be used to ensure correctness of the modified architecture.

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

In this paper, we present an automatic validation framework, driven by an ADL. A novel feature of our approach is the ability to model the pipeline structure and behavior for the processor, coprocessor, as well as the memory subsystem using a graph-based model. Based on this model, we present

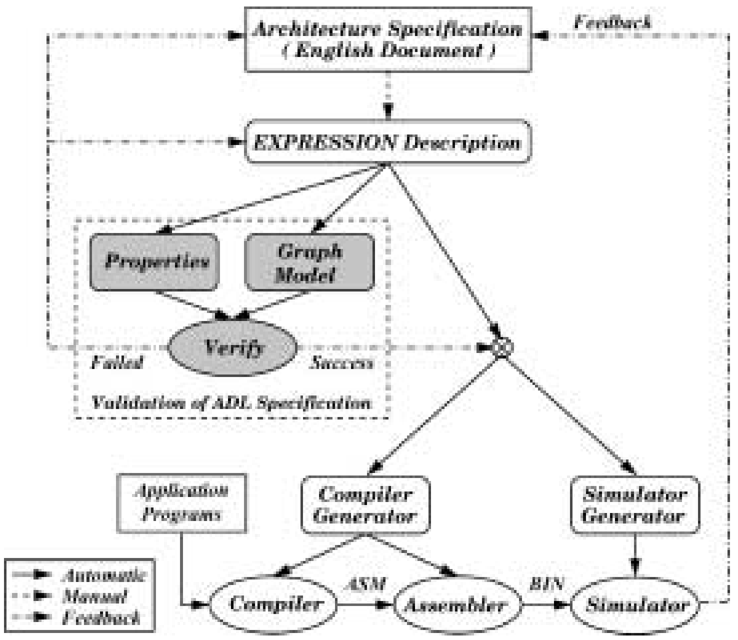

Fig. 3. ADL-driven validation and exploration flow.

algorithms to ensure that the static behavior of the pipeline is well formed by analyzing the structural aspects of the specification. Figure 3 shows the validation flow in our framework. In our ADL driven exploration flow, the designer starts by describing the programmable architecture in an ADL. The graph model of the architecture can be generated automatically from this ADL description. Several properties are applied to ensure that the architecture is well formed. To enable rapid exploration, the software toolkit can be generated from this golden reference model and the feedback can be used to modify the ADL specification of the architecture. We applied our methodology to verify pipeline specification of several realistic architectures from different architectural domains (RISC, DSP, VLIW, and Superscalar) to demonstrate the usefulness of our approach.

The rest of the paper is organized as follows. Sections 2 and 3 present related work addressing ADLs and validation of pipelined processors. Section 4 presents a graph-based modeling of processor, memory, and coprocessor pipelines. Section 5 proposes several properties that must be satisfied for valid pipeline specification. Section 6 illustrates validation of pipeline specifications for several realistic architectures. Finally, Section 7 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 [Freericks 1993] and ISDL [Hadjiyiannis et al. 1997] 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 [Lanneer et al. 1995] 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 such as DSPs, that permit operation parallelism (e.g., Motorola 56K) and VLIW machines with distributed register files (e.g., TI C6X). The retargetable compiler system by Inoue et al. [1998] 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 [Leupers and Marwedel 1998] is an example of an ADL that 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 microcode architecture. The net-list description is used to extract the instruction set [Leupers and Marwedel 1997] 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 [Zivojnovic et al. 1996] is one such ADL whose main characteristic is the operation-level description of the pipeline. RADL [Siska 1998] is an extension of the LISA approach that focuses on explicit support of detailed pipeline behavior to enable generation of production quality cycle-accurate and phase-accurate simulators. FLEXWARE [Paulin et al. 1994] and MDes [Trimaran 1997] 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 [Trimaran 1997] 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, and so on), based on a high-level classification of the information being represented. However, MDes allows only a restricted retargetablility 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, and so on).

The EXPRESSION ADL [Halambi et al. 1999] follows a mixed-level approach (behavioral and structural) to facilitate automatic software toolkit generation, validation, HDL generation, and DSE for a wide range of programmable embedded systems. The ADL captures the structure, behavior, and mapping (between structure and behavior) of the architecture.

3. RELATED WORK

An extensive body of recent work addresses architectural description language driven software toolkit generation and DSE for processor-based embedded systems, as described in Section 2. The ADL specification has also been used to generate hardware implementations for rapid system prototyping [Leupers and Marwedel 1998; Mishra et al. 2003; Schliebusch et al. 2002]. 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 and hardware implementation. 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 [Cyrluk 1993; Sawada and Hunt 1997, 1998; Srivas and Bickford 1990]. However, these approaches require a great deal of user intervention, especially for verifying control intensive designs. Hosabettu [2000] 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 [1994] presented a technique for formally verifying pipelined processor control circuitry. 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 [Skakkebaek et al. 1998; Velev 2000; Velev and Bryant 2000]. The approach of Velev and Bryant [2000] 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 [1998] verified the ARM2 pipelined processor using abstract state machine. Levitt and Olukotun [1997] 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 Aagaard et al. [2001].

Ho et al. [1998] extract controlled token nets from a logic design to perform efficient model checking. Jacobi [2002] 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 [Jhala and McMillan 2001]. There has been a lot a work in the area of module level validation, such as verification of floating-point unit [Ho et al. 1996], and protocol validation, such as verification of cache coherence protocol [Pong and Dubois 1997].

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 that have to be satisfied for the correct pipeline behavior. We apply these properties 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. Currently, we are developing techniques to verify the dynamic behavior by analyzing the instruction flow in the pipeline using a finite state machine based model to validate several architectural properties, such as determinism and execution style, in the presence of hazards and exceptions [Mishra and Dutt 2002; Mishra et al. 2002].

4. ARCHITECTURE PIPELINE MODELING

We present a graph-based modeling of architecture pipelines that captures both the structure and the behavior. The graph model presented here can be derived from a pipeline specification of the architecture described in an ADL, for example, EXPRESSION [Halambi et al. 1999]. This graph model can capture processor, memory, and coprocessor pipelines for wide variety of architectures, namely, RISC, DSP, VLIW, Superscalar, and Hybrid architectures. Note that it is important to capture the memory pipeline along with processor and coprocessor pipelines, since any memory operation exercises both the processor and memory pipeline structures [Mishra et al. 2001]. In this section, we briefly describe how we model the structure and behavior of an architecture. We also model the mapping functions between structure and behavior.

4.1 Structure

The structure of an architecture pipeline is modeled as a graph G_S

$$G_S = (V_S, E_S). \tag{1}$$

V_S denotes a set of components in the architecture. V_S consists of four types of components

$$V_S = V_{unit} \cup V_{storage} \cup V_{port} \cup V_{connection} \tag{2}$$

where V_{unit} is a set of *units* (e.g., ALUs), $V_{storage}$ a set of *storages* (e.g., register files, caches), V_{port} a set of *ports*, and $V_{connection}$ a set of *connections* (e.g., buses). E_S consists of two types of edges

$$E_S = E_{data\_transfer} \cup E_{pipeline} \tag{3}$$

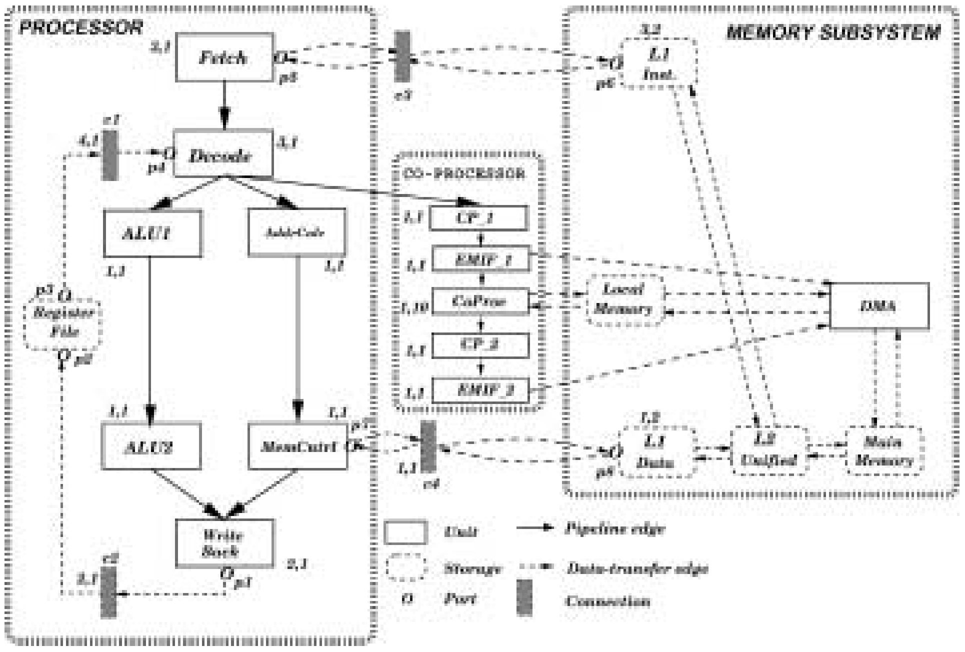

Fig. 4. A structure graph of a simple architecture.

where $E_{data\_transfer}$ is a set of *data-transfer edges* and $E_{pipeline}$ is a set of *pipeline edges*.

$$\begin{aligned} E_{data\_transfer} \subseteq\ & V_{unit} \times V_{port} \\ & \cup\ V_{storage} \times V_{port} \\ & \cup\ V_{port} \times V_{connection} \\ & \cup\ V_{connection} \times V_{port} \\ & \cup\ V_{port} \times V_{unit} \\ & \cup\ V_{port} \times V_{storage} \qquad (4) \\ E_{pipeline} \subseteq\ & V_{unit} \times V_{unit}. \qquad (5) \end{aligned}$$

A data-transfer edge $(v_1, v_2) \in E_{data\_transfer}$ indicates connectivity between two components v_1 and v_2. Data are transferred from one component to another via data-transfer edges. A pipeline edge specifies the ordering of units comprising the pipeline stages (or simply pipe-stages). Intuitively, operations flow from pipe-stages to pipe-stages through pipeline edges. Both pipeline edges and data-transfer edges are unidirectional. Bidirectional data-transfers are modeled using two edges of different directions.

For illustration, we use a simple multi-issue architecture consisting of a processor, a coprocessor, and a memory subsystem. Figure 4 shows the graph-based model of this architecture that can issue up to three operations (an ALU operation, a memory access operation, and a coprocessor operation) per cycle. In the figure, normal boxes denote units, dotted boxes are storages, small circles are ports, shaded boxes are connections, bold edges are pipeline edges, and

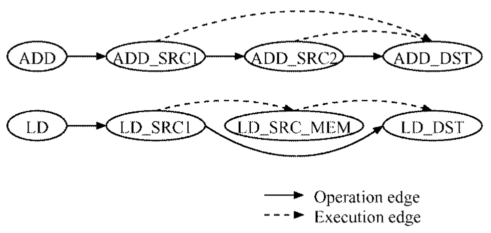

Fig. 5. A fragment of the behavior graph.

dotted edges are data-transfer edges. For ease of illustration, we have shown only few ports and connections. Each component has several attributes. The figure shows only two of them, namely, *capacity* and *timing* for some of the nodes. The capacity denotes the maximum number of operations which the component can handle in a cycle, while the timing denotes the number of cycles taken by the component to execute them. A path from a root node (e.g., Fetch unit) to a leaf node (e.g., WriteBack unit) consisting of units and pipeline edges is called a *pipeline path*. Intuitively, a pipeline path denotes an execution flow in the pipeline taken by an operation. For example, one of the pipeline path is {*Fetch, Decode, ALU1, ALU2, WriteBack*}. A path from a unit to main memory or register file consisting of storages and data-transfer edges is called a *data-transfer path*. For example, {*MemCntrl, L1, L2, MainMemory*} is a data-transfer path. A memory operation traverses different data-transfer paths depending on where it gets the data in the memory. For example, a load operation which is hit in L2 will traverse the path (includes pipeline and data-transfer paths) {*Fetch, Decode, AddrCalc, MemCntrl, L1, L2(hit), L1, MemCntrl, WriteBack*}. Similarly, a coprocessor operation will traverse the path {*Fetch, Decode, CP_1, EMIF_1, CoProc, CP_2, EMIF_2*}. However, in this path we have not shown different data transfers. For example, EMIF_1 sends read request to DMA and DMA writes data in coprocessor local memory which coprocessor uses during computation. The coprocessor writes the result back to local memory and finally EMIF_2 requests DMA to write the result back to main memory.

4.2 Behavior

The behavior of an architecture is a set of operations that can be executed on the architecture. Each operation in turn consists of a set of fields (e.g., opcode, arguments) that specify, at an abstract level, the execution semantics of the operation. We model the behavior as a graph G_B, consisting of nodes V_B and edges E_B.

$$G_B = (V_B, E_B). \tag{6}$$

The nodes represent the fields of each operation, while the edges represent orderings between the fields. The behavior graph G_B is a set of disjointed subgraphs, and each subgraph is called an *operation graph* (or simply an operation). Figure 5 describes a portion of the behavior (consisting of two operation graphs) for the example processor in Figure 4.

Nodes are of two types: V_{opcode} is a set of opcode nodes that represent the opcode (i.e., mnemonic), and $V_{argument}$ is a set of argument nodes that represent argument fields (i.e., source and destination arguments). Each operation graph must have one opcode node. In Figure 5, the ADD and LD nodes are opcode nodes, while the others are argument nodes.

$$V_B = V_{opcode} \cup V_{argument} \tag{7}$$

$$E_B = E_{operation} \cup E_{execution} \tag{8}$$

Edges between the nodes are also of two types. Both types of edges are unidirectional. $E_{operation}$ is a set of operation edges that link the fields of the operation and also specify the syntactical ordering between them. For each operation graph, operation edges must construct a path containing an opcode node. On the other hand, $E_{execution}$ is a set of execution edges that specify the execution ordering between the argument nodes:

$$\begin{aligned} E_{operation} &\subseteq V_{opcode} \times V_{argument} \\ &\quad \cup \; V_{argument} \times V_{argument} \end{aligned} \tag{9}$$

$$E_{execution} \subseteq V_{argument} \times V_{argument}. \tag{10}$$

There must be no cycles consisting of execution edges. In Figure 5, the solid edges represent operation edges while the dotted edges represent execution edges. For the ADD operation, the operation edges specify that the syntactical ordering is opcode followed by ADD_SRC1, ADD_SRC2 and ADD_DST arguments (in that order) and the execution edges specify that the ADD_SRC1 and ADD_SRC2 arguments are executed (i.e., read) before the ADD_DST argument is executed (i.e., written).

4.3 Mapping between Structure and Behavior

An important component of our graph model is a set of functions that correlate the abstract, high-level behavioral model of the architecture to the structural model. We define a set of useful mapping functions that map nodes in the structure to nodes in the behavior (or vice-versa). The *unit-to-opcode* (*opcode-to-unit*) mapping is a bidirectional function that maps unit nodes in the structure to opcode nodes in the behavior. It defines, for each functional unit, the set of operations supported by that unit (and vice versa).

$$f_{unit\text{–}opcode} : V_{unit} \rightarrow V_{opcode} \tag{11}$$

$$f_{opcode\text{–}unit} : V_{opcode} \rightarrow V_{unit} \tag{12}$$

For the example processor in Figure 4, the $f_{unit\text{–}opcode}$ mappings include mappings from Fetch unit to opcodes {*ADD, LD*} , ALU unit to opcode *ADD,* AddrCalc unit to opcode *LD,* and so on.

The *argument-to-storage* (*storage-to-argument*) mapping is a bidirectional function that maps argument nodes in the behavior to storage nodes in the structure. It defines, for each argument of an operation, the storage location that the argument resides in.

$$f_{argument\text{–}storage} : V_{argument} \rightarrow V_{storage} \tag{13}$$

$$f_{storage\text{–}argument} : V_{storage} \rightarrow V_{argument} \tag{14}$$

The $f_{argument\text{–}storage}$ mappings for the LD operation (Figure 5) are mappings from LD_SRC1 to RegisterFile, from LD_SRC_MEM to L1 (Data Memory), and from LD_DST to RegisterFile.

The *port-to-argument* (*argument-to-port*) mapping is a bidirectional function that maps port nodes in the structure to argument nodes in the behavior. It defines, for each port, the arguments that access it.

$$f_{port\text{–}argument} : V_{port} \rightarrow V_{argument} \tag{15}$$

$$f_{argument\text{–}port} : V_{argument} \rightarrow V_{port} \tag{16}$$

The $f_{argument\text{–}port}$ mappings for the LD operation (Figure 5) are mappings from LD_SRC1 to port *p4*, from LD_SRC_MEM to port *p7*, and from LD_DST to *p1*.

Each functional unit (with read or write ports) supports certain data-transfer operations. This can be derived from the above three mapping functions. For example, the *Decode* unit of Figure 4 supports register read (*regRead*) for ADD and LD opcodes; the *MemCntrl* supports data read (*dataRead*) and data write (*dataWrite*) from L1 data cache; the *Fetch* unit supports instruction read (*instRead*) from L1 instruction cache; the *WriteBack* unit supports register write (*regWrite*). Each storage supports certain data-transfer operations. For example, the RegisterFile of Figure 4 supports *regRead* and *regWrite*; L1 data cache supports *dataRead* and *dataWrite*, and so on.

We can generate a graph-model of the architecture from an ADL description that has information regarding architecture's structure, behavior, and the mapping between structure and behavior. We have chosen the EXPRESSION ADL [Halambi et al. 1999] since it captures all the necessary information. We generate automatically the graph model of the architecture pipeline consisting of structure graph, behavior graph, and mapping between them using the modeling techniques described above. Further details on graph model generation can be found in Mishra et al. [2001].

5. ARCHITECTURE PIPELINE VERIFICATION

Based on the graph model presented in the previous section, specification of architecture pipelines written in an ADL can be validated. In this section, we describe some of the properties used in our framework for validating pipeline specification of the architecture.

5.1 Connectedness Property

The connectedness property ensures that each component must be connected to other component(s). As pipeline and data-transfer paths are connected regions of the architecture, this property holds if each component belongs to at least one pipeline or data-transfer path.

$$\forall v_{comp} \in V_S, (\exists G_{PP} \in \mathbf{G_{PP}},\ \text{s.t. } v_{comp} \in G_{PP}) \\ \vee(\exists G_{DP} \in \mathbf{G_{DP}},\ \text{s.t. } v_{comp} \in G_{DP}) \tag{17}$$

where $\mathbf{G_{PP}}$ is a set of pipeline paths and $\mathbf{G_{DP}}$ is a set of data-transfer paths.

Algorithm 1 presents the pseudo code for verifying connectedness property. The algorithm requires the graph model G of the architecture as input. It also

```
Algorithm 1: Verify Connectedness
Inputs: i. Graph model of the architecture $G$
        ii. ListOfUnits: list of units in the graph $G$
        iii. ListOfPorts: list of ports in the graph $G$
        iv. ListOfConnections: list of connections in the graph $G$
        v. ListOfStorages: list of storages in the graph $G$
Outputs: i. True, if the graph model satisfies this property else false.
         ii. In case of failure, print the components which are not connected.
Begin
    Unmark all the entries in all the input lists.
    InsertQ(root, $Q$) /* Put root node of $G$ in queue $Q$ */
    while $Q$ is not empty
        Node $n$ = DeleteQ($Q$) /* Remove the front element of queue $Q$ */
        Mark $n$ as visited in $G$
        case type of node $n$
            unit: Mark $n$ in ListOfUnits
                for each port $p$ in $n$
                    /* Insert $p$ in the rear end of queue $Q$ */
                    if $p$ is not visited, InsertQ(p, $Q$) endif
                endfor
                for each children unit $u$ of $n$
                    if $u$ is not visited, InsertQ(u, $Q$) endif
                endfor
            port: Mark $n$ in ListOfPorts
                for each connection $c$ in $n$
                    if $c$ is not visited InsertQ(c, $Q$) endif
                endfor
                for each storage $s$ associated with $n$
                    if $s$ is not visited InsertQ(s, $Q$) endif
                endfor
                for each unit $u$ associated with $n$
                    if $u$ is not visited InsertQ(u, $Q$)
                endfor
            connection: Mark $n$ in ListOfConnections
                for each port $p$ in $n$
                    if $p$ is not visited InsertQ(p, $Q$) endif
                endfor
            storage: Mark $n$ in ListOfStorages
                for each port $p$ in $n$
                    if $p$ is not visited InsertQ(p, $Q$) endif
                endfor
        endcase
    endwhile
    Return true if all the entries are marked in all of the input lists;
        false otherwise, and print the unmarked components.
End
```

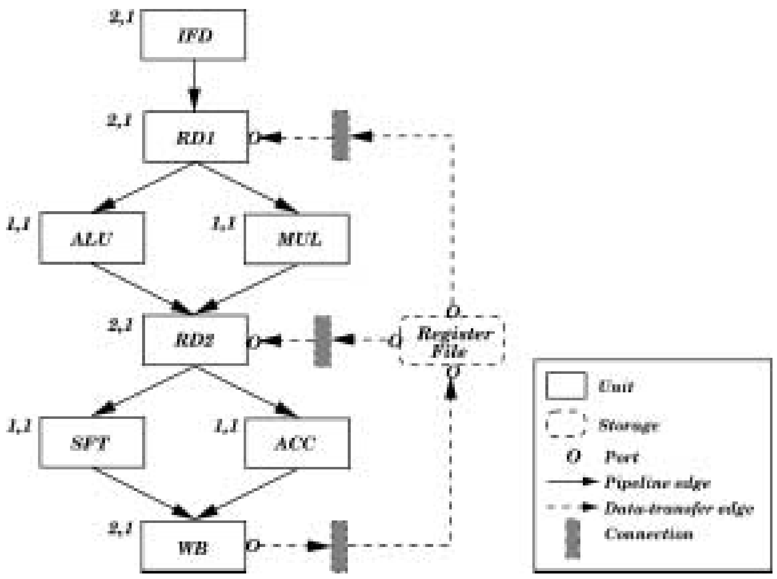

Fig. 6. An example processor with false pipeline paths.

requires all the component (unit, storage, port, and connection) lists as input. The first step is to unmark the entries in all the input lists. Each input list contains all the respective components in the graph. For example, the *ListOfPorts* contains all the ports in the graph *G*. Next, the graph is traversed in breadth-first manner and the visited components are marked. For example, when the unit *u* is visited during traversal, it is marked in *ListOfUnits*. Finally, the algorithm returns true if all the entries are marked in all the input lists. It returns false if there are any unmarked entries in any of the input lists and it prints them.

Each node of the graph is visited only once. The time and space complexity of the algorithm is $O(n)$, where n is the number of nodes in the graph *G*. Each node of the graph can be one of the four components: unit, storage, port, or connection.

5.2 False Pipeline and Data-Transfer Paths

According to the definition of pipeline paths, there may exist pipeline paths that are never activated by any operation. Such pipeline paths are said to be *false*. For example, let us use another architecture shown in Figure 6 that executes two operations: ALU-shift (*alus*) and multiply-accumulate (*mac*). This processor has unit-to-opcode mappings between ALU unit and opcode *alus*, between SFT and *alus*, between MUL and *mac*, and between ACC and *mac*. Also, there are unit-to-opcode mappings between each of {*IFD, RD1, RD2, WB*} and *alus*, and each of {*IFD, RD1, RD2, WB*} and *mac*. This processor has four pipeline paths: {*IFD, RD1, ALU, RD2, SFT, WB*}, {*IFD, RD1, MUL, RD2, ACC, WB*}, {*IFD, RD1, ALU, RD2, ACC, WB*}, and {*IFD, RD1, MUL, RD2, SFT, WB*}. However, the last two pipeline paths cannot be activated by any operation. Therefore, they are

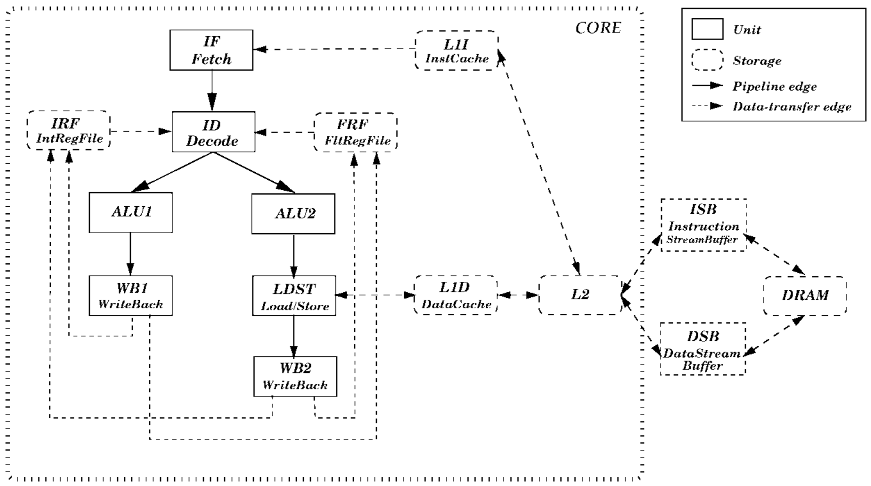

Fig. 7. An example processor with false data-transfer paths.

false pipeline paths. Since these false pipeline paths may become false paths depending on the detailed structure of RD2, they should be detected at a higher level of abstraction to ease the later design phases.

From the view point of systems-on-chip (SOC) architecture exploration, we can view the false pipeline paths as an indication of potential behaviors that are not explicitly defined in the ADL description. This means that further cost, performance, and power optimizations may be possible if new instructions are added to activate the false pipeline paths.

Formally, a pipeline path $G_{PP}(V_{PP}, E_{PP})$ is false if intersection of opcodes supported by the units in the pipeline path is empty.

$$\bigcap_{v_{unit} \in V_{PP}} f_{unit\text{–}opcode}(v_{unit}) = \phi. \tag{18}$$

Similarly, there may exist data-transfer paths in the specification that are never activated by any operation. Such data-transfer paths are said to be *false*. For example, let us use another architecture shown in Figure 7 which has seven possible data-transfer operations: integer-register-read (IregRd), float-register-read (FregRd), integer-register-write (IregWr), float-register-write (FregWr), load-data-from-memory (ldData), load-instruction-from-memory (ldInst), and store-data-in-memory (stData). The *Decode (ID)* unit has mappings for *IregRd* and *FregRd*. There are mappings between each of {*WB1, WB2*} and {*IregWr, FregWr*}, each of {*IF, L1I, ISB*} and *ldInst*, each of {*LDST, L1D, DSB*} and {*ldData, stData*}, and each of {*L2, DRAM*} and {*ldData, stData, ldInst*}. This processor has ten data-transfer paths: {*IRF, ID*}, {*FRF, ID*}, {*WB1, IRF*}, {*WB1, FRF*}, {*WB2, IRF*}, {*WB2, FRF*}, {*IF, L1I, L2, ISB, DRAM*}, {*LDST, L1D, L2, DSB, DRAM*}, {*IF, L1I, L2, DSB, DRAM*}, {*LDST, L1D, L2, ISB, DRAM*} . However, the last two data-transfer paths cannot be activated by any operation. Therefore, they are false data-transfer paths. If *ALU1* supports only floating-point operations, the fourth path ({*WB1, IRF*}) becomes false data-transfer path.

Formally, a data-transfer path $G_{DP}(V_{DP}, E_{DP})$ is false if intersection of data-transfer operations supported by the units and storages ($f_{node-operation}$) in the data-transfer path is empty:

$$\bigcap_{v_{node} \in V_{DP}} f_{node\text{–}operation}(v_{node}) = \phi. \tag{19}$$

Algorithm 2 presents the pseudocode for verifying false pipeline and data-transfer paths. The algorithm requires the graph model G as input. It traverses the graph in depth-first manner along each pipeline and data-transfer path. Each unit u has a list of supported opcodes $SuppOpList_u$. Each node n (unit or storage) also maintains four temporary lists: $OutOpList_n$, $OutDTopList_n$, $InOpList_n$, and $InDTopList_n$. The $OutOpList_n$ is the list of opcodes produced by unit n and sent to its children units. The $OutDTopList_n$ is the list of data-transfer operations produced by node (unit or storage) n and sent to its children storages. The $InOpList_n$ is the list that is used by unit n to copy the $OutOpList_p$, the output list of the recently visited parent p. Similarly, the $InDTopList_n$ is the list that is used by storage n to copy the $OutDTopList_p$, the output list of the recently visited parent p. Each unit n performs intersection of $InOpList_n$ and $SuppOpList_n$ and send the result $OutOpList_n$ to its children units. If $OutOpList_n$ is empty, all the pipeline paths that use the path from n to root (via recently visited parents) are false pipeline paths. A unit with read or write ports computes data-transfer operations using the method described in Section 4.3. A storage computes $OutDTopList_n$ by performing intersection of $SuppDTopList_n$ and the input list $InDTopList_n$. If $OutDTopList_n$ is empty, all the data-transfer paths that use the path from storage n to any unit via recently visited parents are false data-transfer paths. The algorithm returns true if there are no false pipeline or data-transfer paths. It returns false if there are any false pipeline or data-transfer paths and prints them.

If there are n nodes, x pipeline and data-transfer paths in the graph and the number of opcodes supported by the processor is p then the time complexity of this algorithm is $O(x \times n \times (x + p \log p))$ and space complexity is $O(n \times p)$. The opcode list in each node is a sorted list.

5.3 Completeness Property

The completeness property confirms that all operations must be executable. An operation op is executable if there exists a pipeline path $G_{PP}(V_{PP}, E_{PP})$ on which op is executable. An operation op is executable on a pipeline path $G_{PP}(V_{PP}, E_{PP})$ if both conditions (a) and (b) hold.

(a) All units in V_{PP} support the opcode op. More formally, the following condition holds where v_{opcode} is the opcode of the operation op:

$$\forall v_{unit} \in V_{PP}, \quad v_{opcode} \in f_{unit\text{–}opcode}(v_{unit}). \tag{20}$$

(b) There are no conflicting partial ordering of operation arguments and unit ports. Let V be a set of argument nodes of op. There are no conflicting partial ordering of operation arguments and unit ports if, for any two nodes

```
Algorithm 2: Verify False Pipeline and Data-transfer Paths
Input: Graph model of the architecture $G$.
Outputs: i. True, if the graph model satisfies this property else false.
         ii. In case of failure, print the list of false pipeline and data-transfer paths.
Begin
  Push(root, $S$); FalsePPpathList = {}; FalseDPpathList = {} // Push root on stack $S$
  while $S$ is not empty
        Node $n$ = Pop($S$); Mark $n$ as visited /* Pop the top element of stack $S$ */
        case node type of $n$
            unit: if $n$ is the root node
                      $OutOpList_n = SuppOpList_n$ /* Send $OutList_n$ to its children */
                  else
                      // Input list of $n$ is the output list of the recently visited parent $p$
                      $InOpList_n = OutOpList_p$
                      /* Intersection of input list & supported opcode list */
                      $OutOpList_n = SuppOpList_n \cap InOpList_n$
                  endif
                  if $n$ has read or write ports
                      $OutDTopList_n = ComputeDataTransferOps(OutOpList_n)$
                      if $OutDTopList_n$ is empty
                        for all the data-transfer paths fDP from $n$ to any leaf nodes
                            Insert fDP in FalseDPpathList.
                        endfor
                      else
                        for each children storage node st of $n$, Push(st, $S$) endfor
                      endif
                  endif
                  if $OutOpList_n$ is empty
                      Record the pipeline path pp from $n$ by tracing
                      recently visited parents till root
                      for all pipeline paths ppEnd from $n$ to any leaf nodes
                          Append ppEnd to pp to generate false pipeline path fPP.
                          Insert fPP in FalsePPpathList.
                      endfor
                  else
                      for each children unit $u$ of $n$, Push(u, $S$) endfor
                  endif
            storage: $InDTopList_n = OutDTopList_p$
                  $OutDTopList_n = SuppDTopList_n \cap InDTopList_n$
                  if $OutDTopList_n$ is empty
                      Record the data-transfer path dp from $n$ by tracing
                      recently visited parents till any unit
                      for all data-transfer paths dpEnd from $n$ to any leaf nodes
                          Append dpEnd to dp to generate false data-transfer path
                          fDP. Insert fDP in FalseDPpathList.
                      endfor
                  else
                      for each children storage node st of $n$, Push(st, $S$) endfor
                  endif
        endcase
  endwhile
  if FalsePPpathList and FalseDPpathList are empty return true;
  else return false and print FalsePPpathList and FalseDPpathList.
  endif
End
```

$v_1, v_2 \in V$ such that $(v_1, v_2) \in E_{execution}$, all the following conditions hold:

- There exists a data-transfer path from a storage $f_{arg-storage}(v_1)$ to a unit v_{u1} in V_{PP} through a port $f_{arg-port}(v_1)$.
- There exists a data-transfer path from a unit v_{u2} in V_{PP} to a storage $f_{arg-storage}(v_2)$ through a port $f_{arg-port}(v_2)$.
- v_{u1} and v_{u2} are the same unit or there is a path consisting of pipeline edges from v_{u1} to v_{u2}.

For example, let us consider the ADD operation for the processor described in Figures 4 and 5. To satisfy the condition (a), Fetch, Decode, ALU1, ALU2, and WriteBack units must have mappings to the ADD opcode. On the other hand, the condition (b) is satisfied because the structure has data-transfer paths from RegisterFile to Decode and from WriteBack to RegisterFile, and there is a pipeline path from Decode to WriteBack.

Algorithm 3 presents the pseudocode for verifying completeness property. The algorithm requires the graph model *G* and the list of operations *OpList* as input. It traverses the graph in depth-first manner for each operation *op* and identify all the pipeline paths *pp* that support *op*. All the units *n* in the pipeline path should have *op* in their supported opcode list $SuppOpList_n$. The pipeline path *pp* must have units that can read the source operands of *op* and write the destination operands of *op* in correct order. If all the conditions are met, *op* is executable in pipeline path *pp* and *op* is marked in *OpList*. The algorithm returns true if all the entries in *OpList* are marked. It returns false if there are unmarked entries and prints them.

```
Algorithm 3: Verify Completeness
Inputs: i. Graph model G of the architecture.
        ii. The list of operations OpList supported by the architecture.
Outputs: i. True, if the graph model satisfies this property else false.
         ii. In case of failure, print the list of operations that are not executable.
Begin
    for each operation op supported by the architecture
        opSrcList = list of sources in the operation op.
        opDestList = list of destinations in the operation op.
        Push(root, S) /* Put root node of G in stack S */
        while S is not empty
            Node n = Pop(S) /* Remove the top element of S */
            Mark n as visited in G.
            if $op \in SuppOpList_n$ /* op is supported by unit n */
              for each port p of n
                 if p is a read or read-write port
                   for each unmarked source src in opSrcList
                      if src can be read via p, mark src in opSrcList with (p, n)
                      endif
                   endfor
                 endif
                 if p is a write or read-write port
                   for each unmarked destination dest in opDestList
                      if dest can be written via p
                        mark dest in opDestList with (p, n)
                      endif
                   endfor
```

```
                endif
            endfor
            if unit $n$ is a leaf node
               if ((all the sources in $opSrcList$ are marked) and
                   (all the nodes $r$ that read the sources are in expected order)
                   and (all the destinations in $opDestList$ are marked) and
                   (all the nodes $w$ that write the destinations are in expected
                        order)
                   and (all the nodes $r$ and $w$ are in the same pipeline path
                        and $r$ appears before $w$))
                   Mark $op$ in $OpList$ /* this path supports $op$ */
                   break /* one pipeline path is sufficient, exit while loop */
               endif
            else
               for each children unit $u$ of $n$, Push(u, $S$) endfor
            endif
        endwhile
    endfor
    Return $true$ if all the entries in $OpList$ are marked;
            $false$ otherwise, and print the unmarked entries in $OpList$.
End
```

If there are n nodes, x pipeline and data-transfer paths in the graph and the number of opcodes supported by the architecture is p then the time complexity of this algorithm is $O(x \times n \times p \times \log p)$ and space complexity is $O(n \times p)$. The opcode list in each unit is a sorted list.

5.4 Finiteness Property

The finiteness property guarantees the termination of any operation executed through the pipeline. The termination is guaranteed if all pipeline and data-transfer paths except false pipeline and data-transfer paths have finite length and all nodes on the pipeline or data-transfer paths have finite timing.

The length of a pipeline or data-transfer path is defined as the number of stages required to reach the final (leaf) nodes from the root node of the graph.

$$\exists K, \quad \text{s.t.} \quad \forall path \in (\mathbf{G_{PP}}, \mathbf{G_{DP}}), \quad num\_stages(path) < K \tag{21}$$

Here, num_stages is a function that, given a pipeline or data-transfer path, returns the number of stages (i.e., clock cycles) required to execute it. In the presence of cycles in the pipeline path, this function cannot be determined from the structural graph model alone. However, if there are no cycles in the pipeline paths, the termination property is satisfied if the number of nodes in V_S is finite, and each multi-cycle component has finite timing.

Algorithm 4 presents the pseudocode for verifying finiteness property. The algorithm requires the graph model G and the list of operations $OpList$ as input. It traverses the graph in depth-first manner for each operation op and identify all the pipeline paths op–pp that support op. For each opcode it colors different pipeline paths op–pp with different color. A cycle is detected if the same colored node is visited more than once during traversal. The pipeline path op–pp with cycle will be stored in $PathList$. This property is also violated when there are paths that are longer than $MaxPathLength$ or when the execution time needed

```
Algorithm 4: Verify Finiteness
Inputs: i. Graph model G of the architecture.
        ii. The list of operations OpList supported by the architecture.
Outputs: i. True, if the graph model satisfies this property else false.
         ii. In case of failure, prints the list of pipeline and data-transfer paths
             that violates this property.
Begin
    PathList = {};
    for each operation op supported by the architecture
        PathLength = 0;
        Push(< root, PathLength >, S); /* Put root node and PathLength on stack S */
        Unmark all the nodes in graph G;
        ColorCode = 0;
        while S is not empty
            /* Remove the top element (node and PathLength pair) of S */
            < n, PathLength > = Pop(S)
            if op ∈ SuppOpList_n /* op is supported by unit n */
              PathLength = PathLength + 1;
              timing = GetExecutionTime(op, n);
              if ((n is already marked with ColorCode) or
                 (timing is greater than MaxExecutionTime) or
                 (PathLength is greater than MaxPathLength))
                /* Insert path with nodes marked using ColorCode */
                Insert < op, path > pair in PathList
                break; /* exit while loop */
              else
                Mark n with ColorCode
                if unit n is a leaf node
                  ColorCode = ColorCode + 1;
                else
                  for each children node child of n
                      Push(< child, PathLength >, S);
                  endfor
                endif
              endif
            else
              ColorCode = ColorCode + 1;
            endif
        endwhile
    endfor
    Return true if PathList is empty
           false otherwise, and print PathList.
End
```

by *op* in any node in that path is greater than *MaxExecutionTime*. The algorithm returns true if *PathList* is empty. It returns false if there are entries in *PathList* and prints them.

Our finiteness algorithm assumes that there are no cycles in the pipeline. If the cycles are allowed in the pipeline due to the reuse of the resources, our algorithm needs to be modified. Let us assume, a resource is reused by an operation *op* for n_{op} times. We can modify the algorithm to check for "*already marked with ColorCode for* n_{op} *times*" instead of checking "*already marked with ColorCode*" for the operation *op*.

If there are n nodes, x pipeline and data-transfer paths in the graph and the number of opcodes supported by the architecture is p then the time complexity of this algorithm is $O(x \times n \times p \times \log p)$ and space complexity is $O(n \times p)$. The opcode list in each unit is a sorted list.

5.5 Architecture-Specific Properties

The architecture must be well formed based on the original intent of the architectural model. To verify the validity of this property we need to verify several architectural properties. Here we mention some of the architectural properties we verify in our framework.

— The number of operations processed per cycle by a unit cannot be smaller than the total number of operations sent by its parents unless the unit has a reservation station. This is not an error if that specific unit kills certain operations based on certain conditions, for example, killing no operation (NOP).

— There should be a path from an execution unit supporting branch opcodes to program counter (PC) or Fetch unit to ensure that PC is modified in case of branch misprediction.

— The instruction template should match available pipeline bandwidth. However, this is not an error because a machine with n operations in a instruction and m ($> n$) parallel pipeline paths may have many multicycle units. Similarly, the architecture may have m ($< n$) parallel pipeline paths if it has reservation station and instruction fetch timing is large.

— There must be a path from load/store unit to main memory via storage components to ensure that every memory operation is complete.

— The address space used by the processor must be equal to the union of address spaces covered by memory subsystem (SRAM, cache hierarchies, and so on) to ensure that all the memory accesses can complete.

These are only some of the properties we currently verify in our framework. For every architecture with new architectural features we can easily add and verify new properties for those features.

We first verify finiteness property before applying any other properties in our framework. If there are paths with infinite length and timing, the finiteness algorithm will display the path and exit. Next, we apply the connectedness property followed by the false pipeline and data-transfer path property. The remaining properties can be applied in any order. Algorithm 5 shows how we apply these properties in our framework.

```
Algorithm 5: Verify Architecture Specification
Input: Graph model G of the architecture.
Output: True, if the graph model satisfies all the properties else false.
Begin
      status = VerifyFiniteness (G, G.SupportedOpcodeList);
      if (status == false) {
         Print the paths that violates this property;
         Return false;
      }
```

```
    status = VerifyConnectedness ($G$, $G.ListOfUnits$,... );
    **if** (status == $false$) {
        Print the components that are not connected;
        Return $false$;
    }

    status = VerifyFalsePipelineDataTransferPaths($G$);
    **if** (status == $false$) {
        Print the list of false pipeline and data-transfer paths;
        Return $false$;
    }

    status = VerifyCompleteness ($G$, $G.SupportedOpcodeList$);
    **if** (status == $false$) {
        Print the list of operations that are not executable;
        Return $false$;
    }

    /* Apply other architecture specific properties */

    .................

    Return $true$;
**End**
```

6. EXPERIMENTS

In order to demonstrate the applicability and usefulness of our validation approach, we described a wide range of architectures using the EXPRESSION ADL: MIPS R10K, TI C6x, PowerPC, DLX [Hennessy and Patterson 1990], and ARM that are collectively representative of RISC, DSP, VLIW, and Superscalar architectures. We generated the graph model of each of the architecture pipelines automatically from their ADL description. We implemented each property as a function that operates on this graph. Finally, we applied these properties on the graph model to verify that the specified architecture is well formed. Table I shows the specification validation time for the DLX, MIPS R10K, PowerPC, ARM, and TI C6x architectures on a 333 MHz Sun Ultra-II with 128M RAM. This includes the time to generate the graph model from the ADL specification of the architecture and to apply all the properties on the graph model. The validation time depends on three aspects: number of properties applied, complexity of the structure, and the number of operations supported in the architecture.

In the remainder of this section, we describe our specification validation experiments. First, we describe the validation of the DLX specification in detail. Next, we summarize the incorrect specification errors captured by our framework during design space exploration of different architectures.

6.1 Validation of the DLX Specification

Our framework generated graph model G from the ADL specification of the DLX architecture. Figure 8 shows the simplified graph model of the DLX

Table I. Specification Validation Time for Different Architectures

Architecture	ARM	DLX	TI C6x	PowerPC	MIPS R10K
Validation time (seconds)	0.2	0.1	0.2	0.3	0.5

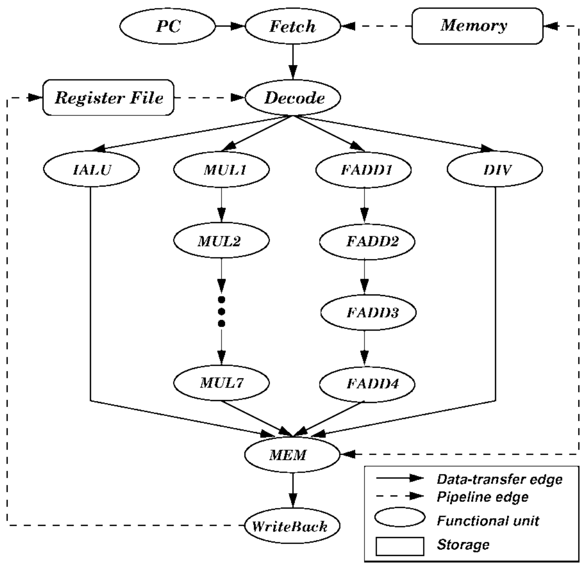

Fig. 8. The graph model of the DLX architecture.

architecture. It does not show ports and connections. The oval (unit) and rectangular (storage) boxes represent nodes. The solid (pipeline) and dotted (data-transfer) lines represent edges.

We applied all the properties (Algorithm 5) on the graph model G. We encountered two kinds of errors, namely, incomplete specification errors and incorrect specification errors. An example of incomplete specification error we uncovered is that the opcode assignment is not done for the fifth stage of the multiplier pipeline. Similarly, an example of the incorrect specification error we found is that only load/store opcodes were mapped for the memory stage (*MEM*). Since all the opcodes pass through memory stage in DLX, it is necessary to map all the opcodes in memory stage as well.

First, the finiteness property is applied on the graph model. It detects a violation for the division operation since the multi-cycle division unit (*DIV*) has undefined latency value. Once the latency for the division operation is defined, the finiteness property is successful. Next, the connectedness property is applied. It detects that the sixth stage of the multiplier unit (*MUL6*) is not connected. Once it is connected properly (from *MUL5* to *MUL6*, and from *MUL6* to *MUL7*), the connectedness property is successful. The false pipeline and data-transfer path detection property is successful. Finally, the completeness property is violated for the multiply operation. This operation is not defined in the *MUL5* unit. As a result, the multiply operation cannot execute in the pipeline. Once this is fixed, the validation of the DLX specification is successful.

6.2 Summary of Property Violations

During design space exploration (DSE) of the architectures, we detected many incorrect specification errors. Here we briefly mention some of the errors captured using our approach.

— We modified the MIPS R10K ADL description to include another load/store unit that supports only store operations. The false data-transfer path property was violated since there was a write connection from the load/store unit to the floating-point register file that will never be used.

— We modified the PowerPC ADL description to have separate L2 cache for instruction and data. Validation determined that there are no paths from L2 instruction cache to main memory. The connection between L2 instruction cache and unified L3 cache is missing.

— We modified the C6x architecture's data memory by adding two SRAM modules with the existing cache hierarchy. The property validation fails due to the fact that the address ranges specified in the SRAMs and cache hierarchy are not disjoint. Moreover, union of these address ranges does not cover the physical address space specified by the processor description.

— We added a coprocessor pipeline to the MIPS R10K architecture that supports vector integer multiplication. This path is reported as a false pipeline path since this opcode was not added in all the units in the path correctly. It also violated the completeness property since the read/write connections to integer register file was missing from the coprocessor pipeline.

— In the R10K architecture we decided to use a coprocessor local memory instead of integer register file for reading operands. We removed the read connections that was used to access the integer register file and added local memory, DMA controller and connections to main memory. The connectedness property is violated for two ports in integer register file. These ports were used by the coprocessor earlier whose connections were deleted but not the ports.

— We modified the PowerPC ADL description by reducing the instruction buffer size from 16 to 4. This generated the violation of architecture-specific property. The fetch unit fetches eight instructions per cycle and decode unit decodes three instructions per cycle, hence there is a potential for instruction loss.

Table II summarizes the errors captured during DSE of architectures. Each column represents one architecture, and each row represents one property. An entry in the table presents the number of violations of that property for the corresponding architecture.[1] The number in brackets next to each architecture

[1]Note that the error numbers will change depending on the number of DSE and type of modifications done each time.

Table II. Summary of Property Violations

	ARM (1)	DLX (2)	C6x (2)	R10K (3)	PowerPC (2)
Connectedness	0	0	1	2	1
False pipeline/data-transfer path	2	5	3	4	2
Completeness	1	2	3	3	2
Architecture-specific	2	4	5	12	6
Finiteness	0	0	0	1	1

represents the number of DSE done for that architecture. Each class of problem is counted only once. For example, the DLX error mentioned above where one of the unit has incorrect specification of the supported opcodes that led to false pipeline path for most of the opcodes, we count that error once instead of using the number of opcodes which violated the property.

Our experiments have demonstrated the utility of our validation approach across a wide range of realistic architectures, and the ability to detect errors in the architectural specification, as well as errors generated through inconsistent modifications to an architecture during DSE.

7. CONCLUSIONS

Architecture description language (ADL) based codesign that supports automatic software toolkit generation is a promising approach to efficient design space exploration (DSE) of system-on-chip (SOC) architectures. The programmable portion of SOCs often includes pipelined processor, memory, and coprocessor cores, whose pipeline structure and behavior are described in the ADL. During architectural design space exploration, each instance of the architecture must be validated to ensure that it is well-formed. Moreover, validation of the specification is essential to ensure that the reference model is golden so that it can be used to uncover bugs in the design.

In this paper, we presented a graph-based modeling of architectures that captures both the structure and the behavior of the processor, memory, and coprocessor pipelines. Based on the model, we proposed several properties that need to be satisfied to ensure that the architecture is well formed. We applied these properties on the graph model of the MIPS R10K, TI C6x, ARM, DLX, and PowerPC architectures, and demonstrated the usefulness of our approach in detecting different types of errors that often appear in the architectural specification during exploration. New properties can be easily defined and applied in our framework. Our ongoing work targets the use of this ADL specification as a golden reference model in architecture validation flow.

ACKNOWLEDGMENTS

This work was partially supported by NSF grants CCR-0203813 and CCR-0205712. We would like to thank Prof. Alex Nicolau, Dr. Peter Grun and Ashok Halambi for their insightful comments. We would like to acknowledge the members of the ACES laboratory for their inputs.

REFERENCES

AAGAARD, M., COOK, B., DAY, N., AND JONES, R. 2001. A framework for microprocessor correctness statements. In *Proceedings of Correct Hardware Design and Verification Methods (CHARME)*, T. Margaria and T. Melham, Ed. Lecture Notes in Computer Science, vol. 2144. Springer-Verlag, Berlin, 433–448.

ARC. *http://www.arccores.com*. ARC Cores.

AXYS. *Axys Design Automation*. http://www.axysdesign.com.

BURCH, J. AND DILL, D. 1994. Automatic verification of pipelined microprocessor control. In *Proceedings of Computer Aided Verification (CAV)*, D. Dill, Ed. Lecture Notes in Computer Science, vol. 818. Springer-Verlag, Berlin, 68–80.

CYRLUK, D. 1993. Microprocessor Verification in PVS: A Methodology and Simple Example. Tech. Rep., SRI-CSL-93-12.

PONG, F. AND DUBOIS, M. 1997. Verification techniques for cache coherence protocols. *ACM Computing Surveys 29*, 1, 82–126.

FREERICKS, M. 1993. The nML machine description formalism. Tech. Rep. TR SM-IMP/DIST/08, TU Berlin CS Dept.

HADJIYIANNIS, G., HANONO, S., AND DEVADAS, S. 1997. ISDL: An instruction set description language for retargetability. In *Proceedings of Design Automation Conference (DAC)*, 299–302.

HALAMBI, A., GRUN, P., GANESH, V., KHARE, A., DUTT, N., AND NICOLAU, A. 1999. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of Design Automation and Test in Europe (DATE)*, 485–490.

HENNESSY, J. AND PATTERSON, D. 1990. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA.

HO, P., HOSKOTE, Y., KAM, T., KHAIRA, M., O'LEARY, J., ZHAO, X., CHEN, Y., AND CLARKE, E. 1996. Verification of a complete floating-point unit using word-level model checking. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, M. Srivas and A. Camilleri, Ed. Lecture Notes in Computer Science, vol. 1166. Springer-Verlag, Berlin, 19–33.

HO, P., ISLES, A., AND KAM, T. 1998. Formal verification of pipeline control using controlled token nets and abstract interpretation. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, 529–536.

HOSABETTU, R. M. 2000. Systematic verification of pipelined microprocessors. Ph.D. thesis, Department of Computer Science, University of Utah.

HUGGINS, J. AND CAMPENHOUT, D. 1998. Specification and verification of pipelining in the ARM2 RISC microprocessor. *ACM Transactions on Design Automation of Electronic Systems 3*, 4, 563–580.

INOUE, A., TOMIYAMA, H., EKO, F., KANBARA, H., AND YASUURA, H. 1998. A programming language for processor based embedded systems. In *Proceedings of Asia Pacific Conference on Hardware Description Languages (APCHDL)*, 89–94.

JACOBI, C. 2002. Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving. In *Proceedings of Computer Aided Verification (CAV)*, E. Brinksma and K. Larsen, Ed. Lecture Notes in Computer Science, vol. 2404. Springer-Verlag, Berlin, 309–323.

JHALA, R. AND MCMILLAN, K. L. 2001. Microarchitecture verification by compositional model checking. In *Proceedings of Computer Aided Verification (CAV)*, G. Berry et al., Ed. Lecture Notes in Computer Science, vol. 2102. Springer-Verlag, Berlin, 396–410.

LANNEER, D., PRAET, J., KIFLI, A., SCHOOFS, K., GEURTS, W., THOEN, F., AND GOOSSENS, G. 1995. CHESS: Retargetable code generation for embedded DSP processors. In *Code Generation for Embedded Processors*. Kluwer Academic, Norwell, MA, 85–102.

LEUPERS, R. AND MARWEDEL, P. 1997. Retargetable generation of code selectors from HDL processor models. In *Proceedings of European Design and Test Conference (EDTC)*, 140–144.

LEUPERS, R. AND MARWEDEL, P. 1998. Retargetable Code generation based on structural processor descriptions. *Design Automation for Embedded Systems 3*, 1.

LEVITT, J. AND OLUKOTUN, K. 1997. Verifying correct pipeline implementation for microprocessors. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, 162–169.

Mishra, P. and Dutt, N. 2002. Modeling and verification of pipelined embedded processors in the presence of hazards and exceptions. In *Proceedings of Distributed and Parallel Embedded Systems (DIPES)*, 81–90.

Mishra, P., Dutt, N., and Nicolau, A. 2001. Architecture description language driven validation of processor, memory, and co-processor pipelines. Tech. Rep. UCI-ICS 01-55, University of California, Irvine.

Mishra, P., Grun, P., Dutt, N., and Nicolau, A. 2001. Processor-memory co-exploration driven by an architectural description language. In *Proceedings of International Conference on VLSI Design*, 70–75.

Mishra, P., Kejariwal, A., and Dutt, N. 2003. Rapid exploration of pipelined processors through automatic generation of synthesizable RTL models. In *Proceedings of Rapid System Prototyping (RSP)*, 226–232.

Mishra, P., Tomiyama, H., Dutt, N., and Nicolau, A. 2002. Automatic verification of in-order execution in microprocessors with fragmented pipelines and multicycle functional units. In *Proceedings of Design Automation and Test in Europe (DATE)*, 36–43.

Paulin, P., Liem, C., May, T., and Sutarwala, S. 1994. FlexWare: A flexible firmware development environment for embedded systems. In *Prof. of Dagstuhl Workshop on Code Generation for Embedded Processors*, 67–84.

Rajesh, V. and Moona, R. 1999. Processor modeling for hardware software codesign. In *Proceedings of International Conference on VLSI Design*, 132–137.

Sawada, J. and Hunt, W. D. 1998. Processor verification with precise exceptions and speculative execution. In *Proceedings of Computer Aided Verification (CAV)*, A. Hu and M. Vardi, Ed. Lecture Notes in Computer Science, vol. 1427. Springer-Verlag, Berlin, 135–146.

Sawada, J. and Hunt J., W.A., 1997. Trace table based approach for pipelined microprocessor verification. In *Proceedings of Computer Aided Verification (CAV)*, O. Grumberg, Ed. Lecture Notes in Computer Science, vol. 1254. Springer-Verlag, Berlin, 364–375.

Schliebusch, O., Hoffmann, A., Nohl, A., Braun, G., and Meyr, H. 2002. Architecture implementation using the machine description language LISA. In *Proceedings of Asia South Pacific Design Automation Conference (ASPDAC)/International Conference on VLSI Design*, 239–244.

Siska, C. 1998. A processor description language supporting retargetable multi-pipeline DSP program development tools. In *Proceedings of International Symposium on System Synthesis (ISSS)*, 31–36.

Skakkebaek, J., Jones, R., and Dill, D. 1998. Formal verification of out-of-order execution using incremental flushing. In *Proceedings of Computer Aided Verification (CAV)*, A. Hu and M. Vardi, Ed. Lecture Notes in Computer Science, vol. 1427. Springer-Verlag, Berlin, 98–109.

Srivas, M. and Bickford, M. 1990. Formal verification of a pipelined microprocessor. In *IEEE Software 7*, 5, 52–64.

Target. *http://www.retarget.com*. Target Compiler Technologies.

Tensilica. *http://www.tensilica.com*. Tensilica Inc.

Trimaran. 1997. *The MDES User Manual*. Trimaran release: http://www.trimaran.org.

Velev, M. and Bryant, R. 2000. Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction. In *Proceedings of Design Automation Conference (DAC)*, 112–117.

Velev, M. N. 2000. Formal verification of VLIW microprocessors with speculative execution. In *Proceedings of Computer Aided Verification (CAV)*, E. Emerson and A. Sistla, Ed. Lecture Notes in Computer Science, vol. 1855. Springer-Verlag, Berlin, 296–311.

Zivojnovic, V., Pees, S., and Meyr, H. 1996. LISA—Machine description language and generic machine model for HW/SW co-design. In *IEEE Workshop on VLSI Signal Processing*, 127–136.

Received May 2002; revised June 2003; accepted August 2003