Automatic Validation of Pipeline Specifications

Prabhat Mishra
pmishra@cecs.uci.edu

Nikil Dutt
dutt@cecs.uci.edu

Alex Nicolau
nicolau@cecs.uci.edu

Architectures and Compilers for Embedded Systems (ACES) Laboratory
Center for Embedded Computer Systems, University of California, Irvine, CA 92697, USA

Abstract

Recent approaches on language-driven Design Space Exploration (DSE) use Architectural Description Languages (ADL) to capture the processor architecture, generate automatically a software toolkit (including compiler, simulator, and assembler) for that processor, and provide feedback to the designer on the quality of the architecture. It is important to verify the ADL description of the processor to ensure the correctness of the software toolkit. We present in this paper an automatic validation framework, driven by an ADL. We present algorithms for automatic validation of ADL specification of the processor pipelines. We applied our methodology to verify several realistic processor cores to demonstrate the usefulness of our approach.

1 Introduction

Embedded systems present a tremendous opportunity to customize designs by exploiting application behavior using customizable processor cores and a variety of memory configurations along with different compiler techniques to meet diverse requirements, such as better performance, low power, smaller area, higher code density etc. However, shrinking time-to-market, coupled with increasingly short product lifetimes create a critical need for rapid exploration and evaluation of candidate SOC architectures. Recent approaches on language-driven Design Space Exploration (DSE) ([1], [2], [8], [9]), use Architectural Description Languages (ADL) to capture the processor architecture, generate automatically a software toolkit (including compiler, simulator, and assembler) for that processor, and provide feedback to the designer on the quality of the architecture. It is important to verify the ADL description of the processor to ensure the correctness of the software toolkit. The benefits of verification are two-fold. First, the process of any specification is error-prone and thus verification techniques can be used to check for correctness and consistency of specification. Second, changes made to the processor during DSE may result in incorrect execution of the system and verification techniques can be used to ensure correctness of the modified architecture.

In this paper we present an automatic validation framework, driven by an ADL. The ADL description captures both the pipeline's structure and behavior for the processor core. Based on this modeling we present algorithms for automatic validation of the processor described in an ADL. We applied our methodology to verify several realistic processor cores from different architectural domains to demonstrate the usefulness of our approach.

The rest of the paper is organized as follows. Section 2 presents related work addressing validation of pipelined processors. Section 3 outlines our approach and the overall flow of our environment. Section 4 briefly describes the EXPRESSION ADL, which is used in our DSE environment. Section 5 proposes our verification technique. The experiments in Section 6 illustrates the usefulness of our approach. Finally, Section 7 concludes the paper.

2 Related Work

An extensive body of recent work addresses Architectural Description Language (ADL) driven software toolkit generation and Design Space Exploration (DSE) for processor-based embedded systems, in both academia: ISDL [2], LISA [8], nML [3], EXPRESSION [1], and industry: ARC [4], Tensilica [6], RADL [12], MDES [9].

While these approaches explicitly capture the processor features to varying degrees and generate automatically a software toolkit for that processor, to our knowledge, there have been very little effort in validating the pipeline specification of the processor described in an ADL. The work of Tomiyama et. al. [7, 10] is a step in this direction. They defined certain properties that need to be verified to ensure that the processor description is well-formed. However, these properties are applicable to simple processor models. Moreover, they do not demonstrate how these properties can be applied in SOC verification during design space exploration.

3 Our Approach

Figure 1 shows the flow in our approach. In our ADL driven design space exploration scenario, the designer starts

by describing the processor core in ADL. Several properties are applied to ensure that the processor core is well-formed. To enable rapid DSE the software toolkit can be generated from the golden ADL specification and the feedback can be used to modify the ADL description of the processor.

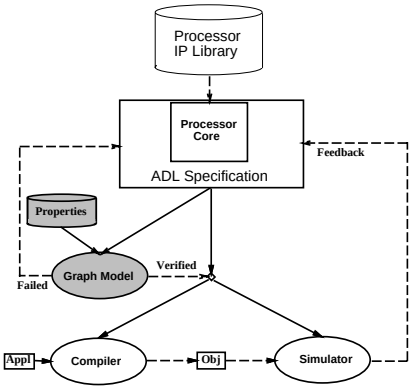

Figure 1. ADL-driven validation flow

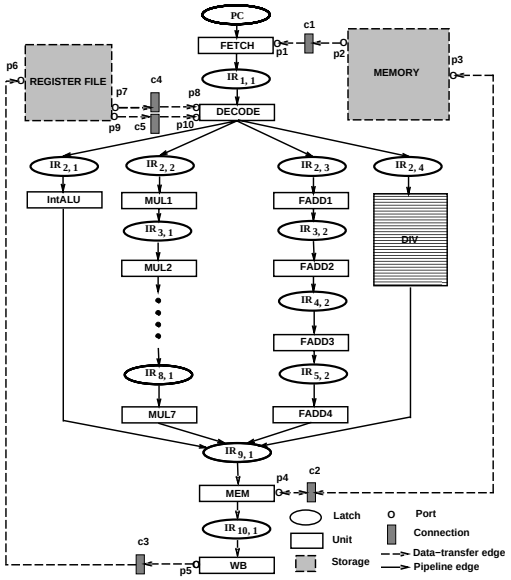

Figure 2. Graph model of simplified DLX processor

4 The EXPRESSION ADL

Our architectural DSE environment uses the EXPRESSION ADL [1] to specify the processor core. In this paper we use EXPRESSION to drive our verification approach; however, the techniques presented in this paper are not specific to any description language, and could be used with any other ADL containing similar information.

EXPRESSION contains an integrated specification of both structure and the behavior of the processor core. The structure of a processor can be viewed as a graph with the components as nodes and the connectivity as the edges of the graph. We consider four types of components: *units* (e.g., ALUs), *storages* (e.g., register files), *ports*, and *connections* (e.g., buses). There are two types of edges: *pipeline edges* and *data transfer edges*. The pipeline edges specify instruction transfer between units whereas data transfer edges specify data transfer between components, typically between units and storages.

Figure 2 shows the graph model of a simplified DLX [5] processor. The oval boxes represent pipeline latches between two units. We call them instruction registers (IR) since they carry instructions. The rectangular boxes represent units, square and shaded boxes represent storages, small circles represent ports and shaded small vertical boxes represent connections, the dotted edges represent pipeline edges, and the solid edges represent pipeline edges. A path from the root node (e.g., FETCH) to leaf node (e.g., WB) consisting of units and pipeline edges is called a pipeline path. For example, *(FETCH, DECODE, IntALU, MEM, WB)* is a pipeline path. A path from a unit to a storage or from a storage to a unit is called a *data-transfer path* if all the edges are data-transfer edges. For example, *(MEM, p4, c2, p3, MEMORY)* is a data-transfer path. Each node in the graph has a list of attributes (optional). The attributes can be any of the following:

- *Latches*: This specifies the list of latches associated with the unit. For example, in Figure 2 the DECODE unit has one parent latch ($IR_{1,1}$) and four children latches ($IR_{2,1}, IR_{2,2}, IR_{2,3}$, and $IR_{2,4}$).
- *Ports*: The list of ports attached to this node. For example, in Figure 2 the DECODE unit has two ports (*p8* and *p10*).
- *Connections*: The list of connections attached to this node. For example, in Figure 2 the *p8* port has the *c4* connection.
- *Opcodes*: The list of opcodes this node supports. For example, in Figure 2 the FADD1 supports all floating-point addition operations.
- *Timing*: For each node it specifies the timing behavior. Timing can be specified on a per-opcode basis, if necessary. For example, the DIV unit in Figure 2 is a multicycle unit with timing 25, i.e., it takes 25 cycles to execute a division operation.
- *Capacity*: The number of operations that can be accepted by this node in a single cycle. For example, in Figure 2 the DECODE unit can accept four operations per cycle.

An instruction is viewed as containing operations that can be executed in parallel. Each instruction contains a list of slots (to be filled with operations) with each slot corresponding to a functional unit. For example, the following

code segment describes an instruction template for decode unit in Figure 2.

```
(INSTR decodeInst
   (WORDLEN 32)
   (SLOTS
       ((OPTYPE DATA_OP) (BITWIDTH 8) (UNIT IntALU))
       ((OPTYPE DATA_OP) (BITWIDTH 8) (UNIT MUL1))
       ((OPTYPE DATA_OP) (BITWIDTH 8) (UNIT FADD1))
       ((OPTYPE DATA_OP) (BITWIDTH 8) (UNIT DIV))
   )
)
```

The behavior is organized into operation groups, with each group containing a set of operations having some common characteristics. For example, the DLX processor has four operation groups: intALU_ops, mult_ops, fadd_ops, and div_ops. Each operation is then described in terms of it's opcode, operands and behavior. Each operand is classified either as source or as destination. Furthermore, each operand has an associated list of register files to which it can be bound. For example, the following code segment describes the add operation where the variable *rf* refers to REGISTER FILE.

```
(OPCODE add
   (OP_TYPE DATA_OP)
   (OPERANDS (SRC1 rf) (SRC2 rf) (DST rf))
   (BEHAVIOR DST = SRC1 + SRC2)
)
```

5 Verification of Processor Pipeline

Based on the processor pipeline model presented in the previous section, the ADL specification of processor pipelines can be verified. In this section, we propose several properties that must be satisfied for valid pipeline specification.

5.1 Connectedness

Each component must be connected to other component(s). The property holds if each component can be accessed by at least one pipeline path or data-transfer path. We briefly outline the algorithm used in our framework to verify this property.

Input: Graph model of the processor *G, ListOfUnits, ListOfPorts, ListOfConnections, ListOfLatches, ListOfStorages*.
Output: *True*, if the graph model satisfies this property else *false*. In case of failure, print the components which are not connected.

1. Unmark all the entries in all the input lists. Each list contains all the respective components in the graph. For example, the *ListOfPorts* contains all the ports in the graph *G*.

2. Traverse each unit *u* of the graph G starting from the root node.

3. Mark the unit *u* in the *ListOfUnits*.

4. For each output latch *l* of *u* mark the appropriate entry in the *ListOfLatches*.

5. For each attached port *p* of *u* mark the appropriate entry in *ListOfPorts*. Traverse all the connections *c* from each of the port *p* and mark the appropriate entry in *ListOfConnections*. For each connection *c* mark the ports *q* on the other side of the connection in *ListOfPorts* (*p* and *q* are the ports on different sides of the same connection *c*).

6. For each children nodes repeat steps 3 to 5.

7. Finally, find out if there are any unmarked entries in any of the lists, viz., *ListOfUnits, ListOfPorts. ListOfConnections, ListOfLatches*, and *ListOfStorages*. If there are no unmarked entries return true else return false and print the unmarked components.

The time and space complexity of the algorithm is $O(n)$, where n is the number of nodes in the graph.

5.2 False Pipeline Path

Each pipeline path must execute at least one valid operation. The pipeline path is not *false* if there are no empty intersection of opcodes supported by the units in the pipeline path, and there are no conflicting partial ordering of operation arguments and unit ports.

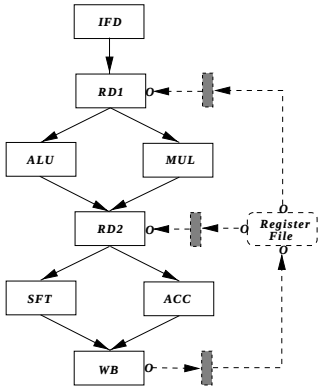

Figure 3. An example processor with false pipeline paths

For example, consider the processor shown in Figure 3, which executes two operations: ALU-shift (*alus*) and multiply-accumulate (*mac*). Each of the unit in {*IFD, RD1, ALU, RD2, SFT, WB*} supports the operation *alus*, and each of the unit in {*IFD, RD1, MUL, RD2, ACC, WB*} supports the operation *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 false pipeline paths. We briefly outline the algorithm used in our framework to verify this property.

Input: Graph model of the processor *G* with each functional unit *u* having it's list of supported opcodes $Supp_u$.
Output: *True*, if the graph model satisfies this property else *false*. In case of failure, it prints the false pipeline paths.

1. Traverse each unit *u* of the graph *G* starting from the root node. The root unit *u* sends $OutList_u$ to it's children, where, $OutList_u = Supp_u$.
2. Each join node (unit) *j* performs union of all the incoming lists from it's parents and forms $InList_j$, for other units (non join nodes) *u* the $InList_u$ is the same as the one sent by it's parent.
3. Each unit *u* computes the $OutList_u$ by performing intersection with $InList_u$ and it's supported opcode list $Supp_u$. It sends the $OutList_u$ to it's children.
4. If $OutList_u$ is NULL, the function returns false path error and enumerates the false path.
5. It returns true if there are no false pipeline paths in the graph *G*.

If there are *n* units in the graph and the number of opcodes supported by the processor is *p* then the time complexity of this algorithm is $O(n \times p)$ and space complexity is $O(n + p)$. The opcode list in each unit is a sorted list.

5.3 Completeness

Each operation must be executable. The completeness property is valid if there is at least one pipeline path supporting the operation. We briefly outline the algorithm used in our framework to verify this property.

Input: Graph model of the processor *G* and the *ListOfOperations* supported by the processor.
Output: *True*, if the graph model satisfies this property else *false*. In case of failure, it prints the operations which violates this property.

1. For each operation *op* with opcode *o* perform the following steps.
2. Identify pipeline paths *p* (from root node to leaf node) which supports the operation *op*. All the units *u* in that path should have the opcode *o* in their supported opcode list $Supp_u$.
3. For each such operation *op* and each pipeline path *p* that supports *op* perform the following:
   - check if some unit, v_{read}, reads all the source operands from storage for the operation *op*
   - check if some unit, v_{write}, writes to storage the destination operand for that operation *op*
   - verify if both v_{read} and v_{write} are in the same pipeline path *p*, and v_{read} is above v_{write} in the pipeline. If yes, mark the operation *op* in *ListOfOperations*.
4. If all the operations are marked in *ListOfOperations* return true else return false and print the operations that are not marked.

The processor may have certain opcodes which are left unused for future expansion. Hence, it is necessary to remove the unused operations from *ListOfOperations* prior to applying this property to avoid incorrect failures. The time complexity of this algorithm is $O(n \times d)$ and space complexity is $O(n + d)$, where *n* is the total number of units in the graph and *d* is the total number of operations in the processor.

5.4 Well-formedness Property

The processor must be well formed. To verify the validity of this property we need to verify several architectural properties:

- The number of operations processed per cycle by a unit cannot be smaller than the total number of operations sent by it's parents if it does not have any reservation station. This property should not be applied to any unit that performs optimizations on input operations e.g., detect and kill NOP operation, constant propagation etc. in which case the number of input operations is reduced.
- There should be a path from an execution unit supporting branch opcodes (branch unit) to PC/Fetch unit to ensure correct branch handling.
- The instruction template (see Section 4) should match available pipeline bandwidth. If instruction template is bigger than available parallel functional units, there is a chance of instruction loss. Similarly, if instruction template is smaller than the available pipeline bandwidth, the resource utilization may not be optimum.

5.5 Finiteness

Termination of the pipeline must be guaranteed and the pipeline must satisfy it's execution semantics. The termination is guaranteed if all the pipeline paths excluding false pipeline paths have finite length and all units in the pipeline paths have finite timing. The length of a pipeline path is defined as the number of stages required to reach the final nodes from the root node of the graph. In presence of cycles,

this cannot be determined from the graph alone. In presence of stalls it is necessary to verify that the stall is resolved in a finite number of cycles. When the stall is dependent on external signals, which come from outside of the processor core, it cannot be verified with the graph model alone. It is a constraint in the design of the blocks which generate such signals. When the stall condition of a unit is dependent on other units, it is necessary to verify that there are no cyclic dependencies for resolving the stall condition.

6 Experiments

We described the MIPS R10K, TI C6x, PowerPC [11], DLX [5], and ARM processors using EXPRESSION ADL [1]. We generated the graph model of each of the processor pipeline automatically from the ADL description. We implemented each property as a function which operates on this graph. Finally, we applied these properties on the graph model to verify that the specified processor is well-formed. The complete validation of each processor specification took less than a second on a 295 MHz Sun Ultra 60 with 1024M RAM.

As expected, we encountered two kinds of errors viz., incomplete specification errors and incorrect specification errors. An example of incomplete specification error we got is that the opcode assignment is not done for the 5th stage of the multiplier pipeline in DLX. Similarly, an example of the incorrect specification error we got is that only load/store opcodes were mapped for the memory stage of the DLX architecture. Since all the opcodes passes through memory stage in DLX, it is necessary to map all the opcodes there.

During design space exploration (DSE) of the architectures we obtained many incorrect specification errors. Here we briefly mention some of the errors captured using our approach. We modified MIPS R10K ADL to include another load/store unit that supports only store operations. Well-formedness property was violated since there was a write connection from load/store unit to floating-point register file, which will never be used. Similarly, we modified PowerPC ADL by reducing the instruction buffer size from 16 to 4. This generated the violation of well-formedness. The fetch unit fetches 8 instructions per cycle and decode unit decodes up to 3 instructions per cycle, hence there is a potential for instruction loss.

7 Summary

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. Furthermore, during architectural design space exploration, each instance of the architecture must be validated to ensure that it is well-formed. In this paper, we present an automatic validation technique for processor pipeline specifications. We propose several properties that need to be satisfied to ensure that the architecture is well-formed. We have applied this methodology successfully on the MIPS R10K, TI C6x, ARM, DLX, and PowerPC processor cores.

Currently, we model and verify the ADL specification of the processor pipeline. We are extending our technique to verify programmable architectures consisting of processor cores, memory subsystem and co-processors.

8 Acknowledgments

This work was partially supported by grants from NSF (MIP-9708067), DARPA (F33615-00-C-1632) and Motorola Corporation. We would like to gratefully acknowledge Hiroyuki Tomiyama, Ashok Halambi, and Peter Grun for their contribution to the pipeline validation work.

References

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

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

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

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

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

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

[7] H. Tomiyama et al. Modeling and verification of processor pipelines in soc design exploration. *HLDVT*, 1999.

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

[9] http://www.trimaran.org. *MDES User Manual*, 1997.

[10] H. Tomiyama et al. Verification of in-order execution in pipelined processors. *HLDVT*, 2000.

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

[12] C. Siska. A processor description language supporting retargetable multi-pipeline dsp program development tools. In *Proc. ISSS*, Dec. 1998.