HDLGen: Architecture Description Language driven HDL Generation for Pipelined Processors

Arun Kejariwal akejariw@cecs.uci.edu
Prabhat Mishra pmishra@cecs.uci.edu
Jonas Astrom astrom@cecs.uci.edu
Nikil Dutt dutt@cecs.uci.edu

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

CECS Technical Report #03-04
Center for Embedded Computer Systems
University of California, Irvine, CA 92697, USA

February, 2003

Abstract

As embedded systems continue to face increasingly higher performance requirements, deeply pipelined processor architectures are being employed to meet desired system performance. System architects critically need modeling techniques to rapidly explore and evaluate candidate architectures based on area, clock frequency, power, and performance constraints. We present an exploration framework for pipelined processors. We use the EXPRESSION Architecture Description Language (ADL) to capture a wide spectrum of processor architectures. The ADL has been used to enable performance driven exploration by generating a software toolkit from the ADL specification. In this paper, we present how to automatically generate synthesizable RTL from the ADL specification using a functional abstraction technique. Automatic generation of RTL enables rapid exploration of candidate architectures under given design constraints such as area, clock frequency, power, and performance. Our exploration results demonstrate the power of reuse in composing heterogeneous architectures using functional abstraction primitives allowing for a reduction in the time for specification and exploration by at least an order of magnitude.

Contents

List of Figures

List of Tables

1 Introduction

Embedded systems present a tremendous opportunity to customize designs by exploiting the application behavior. Shrinking time-to-market, coupled with short product lifetimes create a critical need for rapid exploration and evaluation of candidate System-on-Chip(SOC) architectures. System architects critically need tools, techniques, and methodologies to perform rapid architectural exploration for a given set of applications to meet the diverse requirements, such as better performance, low power, smaller silicon area, higher clock frequency etc.

Recent advances on language driven software toolkit (including compiler and simulator) generation enables performance driven exploration. The software simulator produces profiling data and thus may answer questions concerning the instruction set, the performance of an algorithm and the required size of memory and registers. However, the required silicon area , clock frequency, and power consumption, can only be determined in conjunction with a synthesizable HDL model. Manual or semi-automatic generation of synthesizable HDL model for the architecture is a time consuming process. This can be done only by a set of skilled designers. Furthermore, the interaction among the different teams viz., specification developers, HDL designers, and simulator developers makes rapid architectural exploration infeasible. As a result, system architects rarely have tools or the time to explore architecture alternatives to find the best-in-class solution for the target applications. This situation is very expensive in both time and engineering resources, and has a substantial impact on time-to-market. Without automation and unified development environment, the design process is prone to error and may lead to inconsistencies between hardware and software representations.

Automatic generation of synthesizable HDL design along with software toolkit from a single specification language will be an effective solution for early architectural exploration. The EXPRESSION [1] architecture description language (ADL) was developed for the automatic generation of software toolkit, including compiler, simulator, and assembler. The ADL is used to specify the architecture. The software toolkit is generated automatically from the ADL specification. The application programs are compiled and simulated, and the feedback is used to modify the ADL specification of the architecture. In this paper, we focus on automatic generation of synthesizable HDL along with software toolkit for a wide variety of pipelined architectures.

The contribution of this work is a methodology for automatic generation of synthesizable HDL models from a specification language to enable rapid exploration of pipelined architectures. The existing approaches are either semi-automatic (expects designers to write datapath components manually) or covers a restricted set of architectures. However, none of these approaches are able to capture a wide spectrum of processor features present in DSP, VLIW, EPIC and Superscalar processors, and generate synthesizable RTL from the ADL specification. The main bottleneck has been the lack of an abstraction (covering a diverse set of architectural features) that permits the reuse of the primitives to compose the heterogeneous architectures. We are able to generate synthesizable HDL description for a wide range of pipelined architectures using a functional abstraction technique. Due to our single specification driven exploration approach the hardware and software representations are consistent.

The rest of the paper is organized as follows. Section 2 briefly describes the EXPRESSION ADL. Section 3 presents related work addressing language driven HDL generation approaches. Section 4 presents the functional abstraction technique that permits reuse of abstraction primitives to compose heterogeneous architectures. Section 5 outlines our approach and the overall flow of our ADL driven exploration environment. The synthesizable HDL generation method is described in Section 6. Section 8 presents preliminary experiments using our system. Section 9 concludes the paper.

2 The EXPRESSION ADL

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 (processor, co-processor, and memory subsystem). The EXPRESSION ADL has been used to generate compiler, simulator, and assembler for the TI C6x [25], PowerPC [5], ARM [26], Hitachi SH3 [27], ST100 [28], Sun UltraSparc-III [7], and MIPS R10K [6] architectures. The ADL can be used to perform top-down validation of programmable embedded systems ([15], [20], [17]). In this paper, we demonstrate the capability of the framework to perform rapid exploration through automatic generation of synthesizable RTL. We briefly describe the key aspects of the EXPRESSION ADL in this section. The complete reference of the language is provided in [1].

The EXPRESSION ADL captures the structure, behavior, and mapping (between structure and behavior) of the programmable architecture as shown in Figure 1

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 via pipeline latches, whereas data transfer edges specify data transfer between components, typically between units and storages or between two storages. Each component has a list of at-

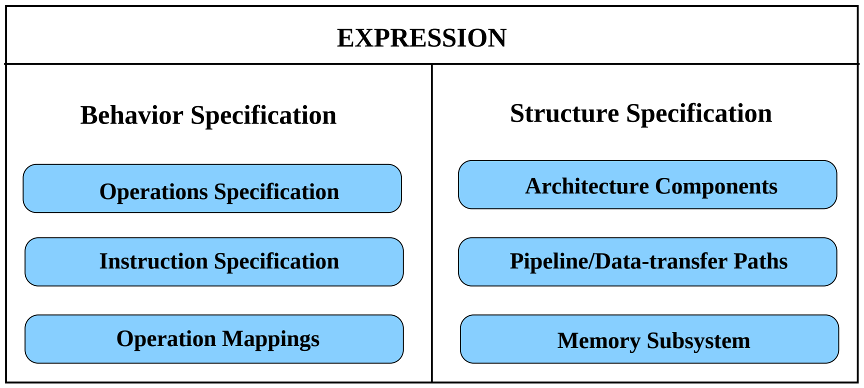

Figure 1. The EXPRESSION ADL

tributes. For example, a functional unit will have information regarding latches, ports, connections, opcodes, timing, capacity etc. For example, the *timing* attribute will have information regarding the timing of each opcode supported by the unit.

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, behavior, and instruction format. Each operand is classified either as source or as destination. Furthermore, each operand is associated with a type that describes the type and size of data it contains. The instruction format describes the fields of the operation in both binary and assembly. The binary format for the following *add* operation has opcode (0101) field from 26th bit to 29th bit.

```
(OPCODE add
  (OP_TYPE DATA_OP)
  (OPERANDS (SRC1 int_rf) (SRC2 int_32) (DST int_rf))
  (BEHAVIOR DST = SRC1 + SRC2)
  (FORMAT cond(31-30) 0101 dest(25-21) src1(20-16) .... )
  .......
)
```

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). For example, the operation *add* is mapped to *ALU* unit.

3 Related Work

Two major approaches to synthesizable HDL generation have been proposed. The first one is a generic parameterized processor core based approach. These cores are bound to a single processor template whose tools and architecture can be modified to a certain degree. Another approach is based on processor specification languages. The language based approaches permit specification of the processor at the expense of restrictions on the quality and/or availability of the tools.

Examples for processor template based approaches are Xtensa [24], Jazz [8], and PEAS [12]. Xtensa [24] is a scalable RISC processor core. Configuration options include the width of the register set, caches, memories etc. New functional units and instructions can be added using the Tensilica Instruction Language (TIE). A synthesizable hardware model along with software toolkit can be generated for this class of architectures. Improv's Jazz [8] processor is a VLIW processor that permits the modeling and simulation of a system consisting of multiple processors, memories, and peripherals. It allows modifications of data width, number of registers, and the depth of hardware task queue. It is also possible to add functional units and add custom functionality in Verilog. The PICO-NPA [22] system automatically synthesizes nonprogrammable accelerators to be used as co-processors for functions expressed as loop nests in C. PEAS [12] is a GUI based hardware/software codesign framework. It generates HDL code along with software toolkit. It has support for several architecture types and a library of configurable resources. Instruction set and micro-operations are separately described.

Processor description language based HDL generation frameworks can be divided into environments based on the type of information the language can capture. nML [10] and ISDL [4] languages capture the instruction set (IS) of the processor. In nML, the processor IS is described as an attributed grammar with the derivations reflecting the set of legal instructions. The architectural scope is limited to DSPs and ASIPs for nML based specification. In ISDL, constraints on parallelism are explicitly specified through illegal operation groupings. As the generation of functional units is the result of an analysis and optimization process of the HDL generator HGEN, the designer has only indirect influence to the generated HDL model. Itoh et al. [11] have proposed a micro-operation description based synthesizable HDL generation. The structure of the processor is derived by analyzing the micro-operation descriptions. It can handle a very simple processor model with no hardware interlock mechanism or multi-cycle operations. It relies on compiler to insert necessary NOP instructions. As a result, it does not support instructions related to interrupt, cache control, co-processor etc.

MIMOLA [21] captures the structure of the processor wherein the net-list of the target processor is described in a HDL like language. Extracting the instruction set from the structure may be difficult for complicated instructions.

More recently, languages that capture both the structure and the behavior (instruction set) of the processor, as well as detailed pipeline information have been proposed. LISA [29] is one such language that captures operation-level description of the pipeline. The synthesizable HDL generation approach based on LISA language ([2], [13], [14]) is closest to our approach. The LISA machine description provides information consisting of model components for memory, resource, instruction set, behavior, timing, and micro-architecture. It can model only DSP and VLIW architectures. It generates HDL model of the processor's control path and the structure of the pipeline. However, the designer has to manually implement the datapath components. A major problem here is verification since operations have to be described and maintained twice - on the one hand in the LISA model and on the other hand in the HDL model (hand written datapath) of the target architecture. Due to the need of manual intervention, this method is not suitable for rapid design space exploration.

The methodology we present in this paper combines the advantages of the processor template based environments and the language based specifications. In fact, we have taken template based design one step ahead using our functional abstraction technique. Thus unlike previous approaches, we are able to efficiently explore a wide range of pipelined architectures exhibiting heterogeneous architectural styles, as well as the memory subsystems.

4 Functional Abstraction

While contemporary ADLs can effectively capture one class of architecture, they are typically unable to capture a wide spectrum of processor and memory features present in DSP, VLIW, EPIC and Superscalar processors. The main bottleneck has been the lack of an abstraction (covering a diverse set of architectural features) that permits the reuse of the primitives to compose the heterogeneous architectures. In this section, we briefly describe the functional abstraction needed to capture such wide variety of programmable architectures. The complete reference is available in [16].

The notion of functional abstraction comes from a simple observation: different architectures may use the same functional unit (e.g., fetch) with different parameters, the same functionality (e.g., operand read) in different functional unit, or new architectural features. The first difference can be eliminated by defining generic functions with appropriate parameters. The second difference can be eliminated by defining generic sub-functions, which can be used by different architectures at different points in time. The last one is difficult to alleviate since it is new, unless this new functionality can be composed of existing sub-functions (e.g., *multiply-accumulate* operation by combining *multiply* and *add* operations). We have defined the necessary generic functions, sub-functions and computational environment needed to capture a wide variety of processor and memory features. We first explain the functional abstraction needed to capture the structure and behavior of the processor and memory subsystem, then we discuss the issues related to defining generic controller functionality.

```
FetchUnit ( # of read/cycle, res-station size, ....)
{
   address = ReadPC();
   instructions = ReadInstMemory(address, n);
   WriteToReservationStation(instructions, n);
   outInst = ReadFromReservationStation(m);
   WriteLatch(decode_latch, outInst);

   pred = QueryPredictor(address);
   if pred {
      nextPC = QueryBTB(address);
      SetPC(nextPC);
   } else
      IncrementPC(x);
}
```

Figure 2. A Fetch Unit Example

We capture the structure of each functional unit using parameterized functions. For example, the fetch unit functionality contains several parameters, viz., number of operations read per cycle, number of operations written per cycle, reservation station size (size==0 implies no reservation station), branch prediction scheme, number of read ports, number of write ports etc. Figure 2 shows a specific example of a fetch unit described using sub-functions. Each sub-function is defined using appropriate parameters. For example, *ReadInstMemory* reads *n* operations from instruction cache using current PC address (returned by *ReadPC*) and writes them to the reservation station. The notion of generic sub-function allows the flexibility of specifying the system in finer detail. It also allows reuse of the components. Furthermore, these components can be pre-verified. Thus the task of verification will reduce to mainly performing interface verification at all levels.

The behavior of a generic processor is captured through the definition of opcodes. Each opcode is defined as a function, with a generic set of parameters, which performs the intended functionality. The parameter list includes source and destination operands, necessary control and data type information. We have defined common sub-functions e.g., ADD, SUB, SHIFT etc. The opcode functions may use one or more sub-functions. For example, the MAC (multiply and accumulate) uses two sub-functions (ADD and MUL) as shown in Figure 3.

```
ADD (src1, src2) {
  return (src1 + src2);
}
```

```
MUL (src1, src2) {
  return (src1 * src2);
}
```

```
MAC (src1, src2, src3) {
   return ( ADD( MUL(src1, src2), src3) );
}
```

Figure 3. Modeling of MAC operation

Each type of memory module viz., SRAM, cache, DRAM, SDRAM, stream buffer, victim cache etc., is modeled using a function with appropriate parameters. For example, the cache function has parameters: cache size, line size, associativity, word size, replacement policy, write policy, read/write access times etc. These functions also have parameters for specifying pipelining, parallelism, access modes (normal read, page mode read, burst read etc.) etc.

The controller is maintained by using a generic control table. The number of rows in the table is equal to the number of pipeline stages in the architecture. The number of columns is equal to the maximum number of parallel units present in any pipeline stage. Each entry in the control table corresponds to one particular unit in the architecture. It also contains information specific to that unit e.g., busy bit (BB), stall bit (SB), list of children, list of parents, opcodes supported etc. This table is generated from the ADL specification. The control table captures all the necessary details to perform necessary stalling and flushing of the pipelines in the presence of branches, hazards, and exceptions.

We have also developed a generic scheme for defining interrupts, exceptions, interrupt handler, DMA, co-processor, and external interface. The detailed description of generic abstractions for all of the microarchitectural components are too long to describe in this section, and can be found in [16].

5 Our Approach

Figure 4 shows our ADL driven architecture exploration framework. System designers have initial ideas about the programmable architecture (processor, coprocessor, memory subsystem) for the given set of applications. They develop architecture specification document based on their expertise and available prototypes.

The first step is to specify the architecture in EXPRESSION ADL. It is necessary to validate the ADL specification to ensure that the architecture is well-formed ([15], [18], [20]). It guarantees the correctness of the generated software toolkit and the HDL implementation. The software toolkit, including compiler, assembler, and simulator, is generated automatically from the ADL specification. The application program is compiled and simulated to generate performance numbers. The simulator produces profiling data and thus may answer questions concerning the instruction set, the performance of an algorithm and the required size of memory and registers. The required silicon area, clock frequency, or power consumption can only be determined by generating a synthesizable HDL model.

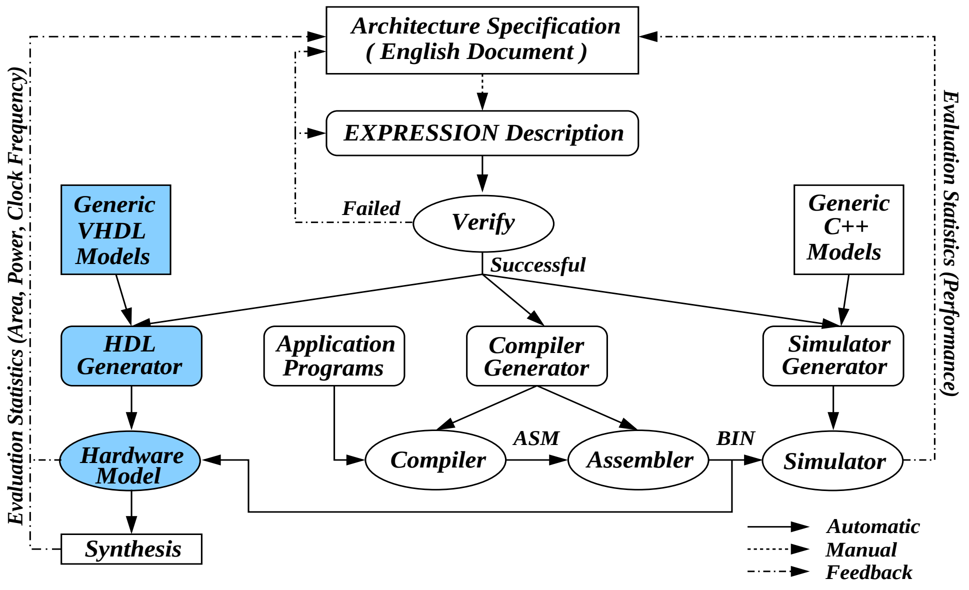

Figure 4. Architecture Exploration Framework

In this paper, we present automatic generation of synthesizable HDL models (shown shaded in Figure 4) from the ADL specification. We have developed VHDL models for each generic function and sub-functions as described in Section 4. Our *HDL generator* is capable of composing heterogeneous architectures using functional abstraction primitives. The generated hardware model (VHDL Description) is synthesized using Synopsys Design Compiler [23] to generate evaluation statistics: area, clock frequency, and power consumption. The hardware model can also be used to generate performance numbers. We perform this step to validate the generated hardware model against the simulator.

6 Synthesizable HDL Generation

We have already demonstrated the power of functional abstraction in generating simulation models for a wide variety of architectures allowing for a reduction in the time for specification and exploration by at least an order of magnitude [19]. In this paper we have used the functional abstraction technique to automatically generate synthesizable VHDL models from the ADL specification. In fact, there is a direct relationship between generating a simulator and a hardware model: the synthesizable VHDL model is itself a simulator.

The generated HDL description consists of four major parts: instruction fetch, instruction decoder, data-path, and control logic. We have implemented all the generic functions and subfunctions (as described in Section 4) using VHDL. The following sections briefly describe the above steps.

6.1 Instruction Fetch

We have implemented a parameterized instruction fetch unit. The fetch bandwidth between the fetch unit and the instruction memory is specified using EXPRESSION. A Pre-fetch buffer is modeled to facilitate large availability of instructions during a branch prediction phase for early execution. The fetch unit feeds instructions to the decoder from the pre-fetch queue. However, it should be noted that in case the pre-fetch queue is empty, the newly fetched instructions from the instruction memory are directed to the decoder rather than the pre-fetch queue to performance penalty.

6.2 Instruction Decoder

We have implemented a generic instruction decoder that uses information regarding individual instruction format and opcode mapping for each functional unit to decode a given instruction *inst*. The instruction format information is available in operations section of the EXPRESSION ADL. The decoder extracts information regarding opcode, operands etc. from input instruction *inst* using the instruction format. The mapping section of the EXPRESSION captures the information regarding the mapping of opcodes to the functional units. The decoder uses this information to perform/initiate necessary functions (e.g., operand read) and decide where (pipeline latch) to send the instruction. Based on the register availability information, the decoder carries out register renaming, hence supporting multiple instruction issue.

6.3 Data Path

The implementation of datapath consists of two parts. First, compose each component in the structure. Second, instantiate components (e.g., fetch, decode, ALU, LdSt, writeback, branch, caches, register files, memories etc.) and establish connectivity using appropriate number of pipeline latches, ports, and connections using the structural information available in the ADL.

To compose each component in the structure we use the information available in the ADL regarding the functionality of the component and its parameters. For example, to compose an execution unit, it is necessary to instantiate all the opcode functionalities (e.g, ADD, SUB etc. for an ALU) supported by that execution unit. Also, if the execution unit is supposed to read the operands then appropriate number of operand read functionalities needs to be instantiated unless the same read functionality can be shared using multiplexors. Similarly, if this execute unit is supposed to write the data back to register file, the functionality for writing the result needs to be instantiated. The actual implementation of an execute unit might contain many more functionalities e.g., read latch, write latch, insert/delete/modify reservation station (if applicable).

6.4 Control Logic

The controller is implemented in two parts. First, it generates a centralized controller (using generic controller function with appropriate parameters) that maintains the information regarding each functional unit such as busy, stalled etc. It also computes hazard information based on the list of instructions currently in the pipeline. Based on these bits and the information available in the ADL it stalls/flushes necessary units in the pipeline. Second, a local controller is maintained at each functional unit in the pipeline. This local controller generates certain control signals and sets necessary bits based on input instruction. For example, the local controller in an execute unit will activate the add operation if the opcode is *add*, or it will set the busy bit in case of a multi-cycle operation.

7 Synthesis-based Optimizations

In this section we briefly describe three synthesis-based optimizations that enable area, power, and performance improvement.

7.1 Resource Sharing

In minimum area applications it is necessary to maximize the resource sharing under given timing constraints. Consider a move operation that is implemented using a bus, and load and store operations that are mutually exclusive with the move. Additionally, the move operation resides in a different field than the load and store operations. A naive scheme would generate additional data-paths to handle the load and store operations even though it is possible to implement these with the same bus that implements the move.

Here, we carry out resource constrained scheduling where the designer specifies the number of units available of each operator type. All the ready operations are issued as soon as the required functional unit (FU) is made available. This avoids redundant instantiations of an operator type in different modules of the system. However, care should be taken to check performance penalty due to resource sharing i.e. enough FUs should be made available to extract maximum parallelism amongst the different modules.

7.2 Bit-width Optimizations

Conventional synthesis performs the allocation of heterogeneous specifications, those formed by operations of different types and widths, by binding operations to functional units of their same type and width. Such implementations incur hardware overhead. This HW overhead could be reduced by jointly allocating all compatible operations (those with a common operative kernel) independently of their widths. This definition is transitive and considers trivial cases like the compatibility between additions and subtractions, and more complex ones like the compatibility between additions and multiplications.

$A_{20\text{ bits}} = B_{10\text{ bits}} \times C_{10\text{ bits}}$ $D_{10\text{ bits}} = E_{6\text{ bits}} \times F_{4\text{ bits}}$

$G_{20\text{ bits}} = H_{16\text{ bits}} \times I_{4\text{ bits}}$ $J_{14\text{ bits}} = K_{8\text{ bits}} \times L_{6\text{ bits}}$

Figure 5. A Multiplication Example

Figure 5 shows four multiplications using different bit-width operands. Conventional synthesis will instantiate four multipliers for the example in Figure 5. However, *bit-width enabled* synthesis will instantiate only two multipliers as shown in Figure 6. Such, optimizations inherently reduce the number of *bit-value transitions*, thus reducing the power requirements of the design.

7.3 Communication Overhead

One of the key issues in hardware/software codesign is the generation of efficient interprocess communication interfaces by determining structure and type of communication between different components of a digital system. In obtaining full benefit of high performance components, the problem of communication bottlenecks is one of the main obstacles. Most of the computationally intensive algorithms have to deal with an immense amount of data. The generation of application specific communication structures is a time consuming and error prone task in real-time embedded system design. Hence, ADL driven communication synthesis provides a good platform to evaluate

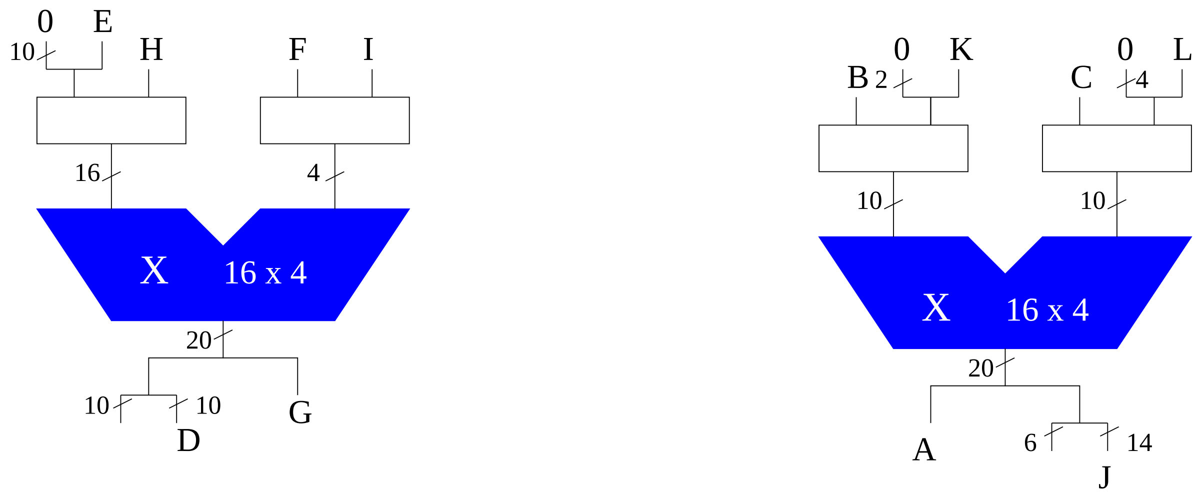

Figure 6. An example illustrating potential bit-width optimizations

and validate the communication overhead in large class of architectures. As a future work, we plan to explore the same using the EXPRESSION ADL.

8 Experiments

We performed architectural design space exploration by varying different architectural features, achieved by reusing the abstraction primitives with appropriate parameters. In this section, we illustrate the usefulness of our approach by generating synthesizable HDL description and performing rapid exploration of the DLX architecture [9].

We have chosen DLX processor for two reasons. First, it has been well studied in academia and there are HDL implementations available for the DLX processor that we can use for comparison purposes. Second, it has many interesting features viz., fragmented pipelines, multi-cycle units etc., that are representative of many commercial pipelined processor architectures such as TI C6x [25], PowerPC [5], and MIPS R10K [6].

8.1 Experimental Setup

Figure 7 shows the DLX processor pipeline that we have captured in our framework. The DLX architecture has five pipeline stages: fetch, decode, execute, memory, and writeback. The *execute* stage has four parallel paths: integer ALU, 7 stage multiplier, four stage floating-point adder, and multi-cycle divider.

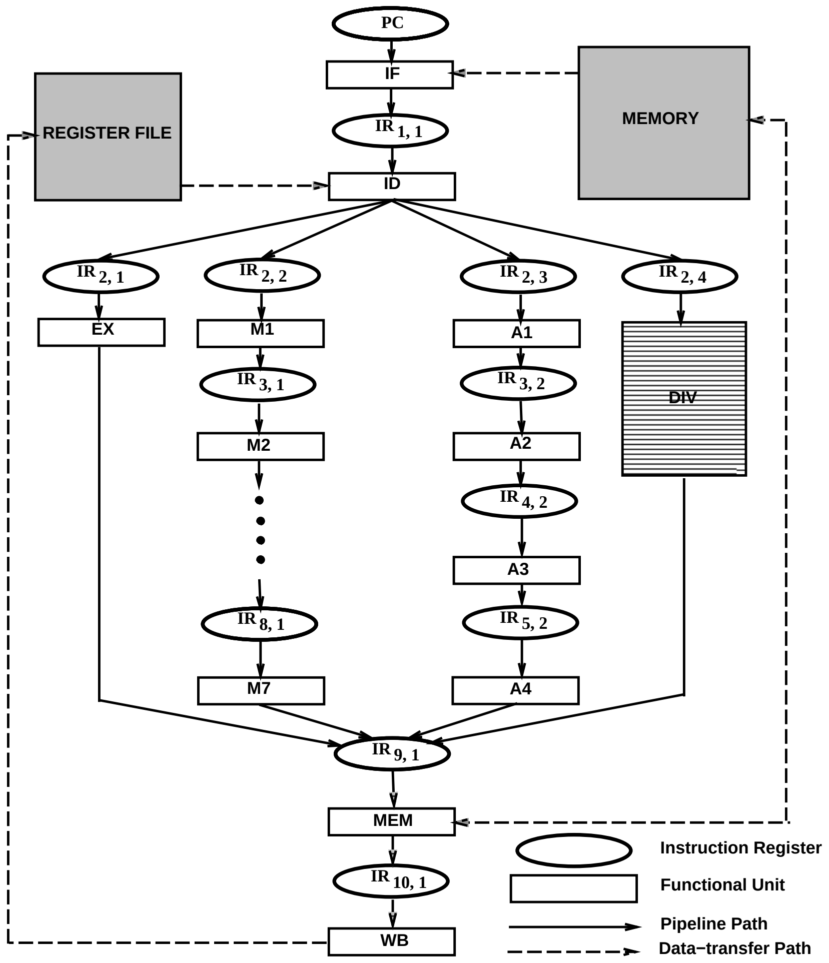

Figure 7. The DLX Processor

The EXPRESSION ADL captures the structure and behavior of the DLX architecture. Synthesizable HDL models are generated automatically from this specification. We have not done any manual changes to the generated HDL models to enhance results. The generated HDL model is validated against the generated simulator using the application programs from DSP and Multimedia domains. We have used Synopsys Design Compiler [23] to synthesize the generated HDL description using LSI 10K technology libraries and to obtain measures in terms of power consumption, clock frequency, and chip area. The following lists the commands used to obtain the performance and power metrics using the Synopsys Design Compiler.

```
/* For high effort compilation */
compile  -map_effort high

/* Synopsys DA command for obtaining various metrics */
report_timing -max_paths 3  -nosplit -path full > <modulename>_timing.report
report_qor > <modulename>_qor.report
report_area > <modulename>_area.report
report_power -analysis_effort high > <modulename>_power.report
```

Our *functional abstraction* approach helped us to restrict the size of individual modules, which is critical for synthesis. Memory overflow like problems were observed for large designs. It helped to fine tune our design for better results.

8.2 Results

We have performed extensive architectural explorations by varying different architectural features. In this section we present three exploration experiments starting with a simple DLX architecture.

We captured the RISC version of the DLX (say *RISC-DLX*) architecture in the ADL and generated HDL description. We compare the quality of the generated HDL code with the results published by Itoh et al.[11] (say *PEAS-DLX*), which is synthesized using VSC753d (CMOS 0.5 μm) technology library. Table 1 presents the comparative results between *PEAS-DLX* [11] and *VLIW-DLX* (generated by our framework). The second column lists the number of words in the specification of both designs. The third column presents the number of words in the generated HDL code. The fourth and fifth columns describe the area (gate count) and clock frequency respectively. The power consumption for our hardware model is 52.4 mW.

	Spec (words)	HDL Code (words)	Area (gates)	Speed (MHz)
RISC-DLX	2063	6612	118 K	33
PEAS-DLX	1196	6259	105 K	5.3

Table 1. Synthesis Results: RISC-DLX vs PEAS-DLX

Next, we modified this *RISC-DLX* model to add several architectural features from VLIW and Superscalar domains that would not be possible with any existing ADL based framework.

We modified the ADL specification of the DLX (say *VLIW-DLX*) to add support for interlocking, stalling, flushing, and multi-cycle operations. We have obtained area (185 K gates) and power (70 mW) numbers for this model.

Finally, we modified the ADL specification to add superscalar features (say *Superscalar-DLX*) and generated the HDL description. We have obtained the hand written version of the Superscalar DLX (say *Darmstadt-DLX*) processor from the repository of Darmstadt University of Technology [3]. We made necessary modifications to make it synthesizable using Synopsys Design Compiler. Table 2 presents the comparative results between *Darmstadt-DLX* (modified) and *Superscalar-DLX* (generated by our framework). The second, third and fourth columns describe the area (gate count), clock frequency, and power consumption respectively.

	Area (gates)	Speed (MHz)	Power (mW)
Superscalar-DLX	239 K	20	108.27
Darmstadt-DLX	198 K	27.7	71

Table 2. Hardware Synthesis Results

Our generated design (without any manual intervention) is 20-40% off in terms of area, power, and clock speed. Indeed, these are reasonable ranges for rapid system prototyping and exploration. Each iteration in our exploration framework is in the order of hours to days depending on the amount of modification needed in the ADL and the synthesis time. However, each iteration will be in the order of weeks to months for manual or semi-automatic development of HDL models. The reduction of HDL generation time is at least an order of magnitude.

We have analyzed our results and observed that the execution units are consuming 50-60% of the total area and power. This is due to the fact that we have not considered the optimization and resource sharing issues of our data path components yet.

9 Summary

We have presented a synthesizable HDL generation method for pipelined processors from an ADL specification. The EXPRESSION ADL captures the structure and the behavior of the architecture. The synthesizable HDL description is generated automatically from the ADL specification using the functional abstraction technique. The generated hardware model is validated against the generated simulator. The synthesis of the generated HDL model is performed to generate evaluation statistics such as chip area, clock frequency, and power consumption. The feasibility of our technique is confirmed through experiments. The result shows that a wide varieties of processor features can be explored in hours to days – an order of magnitude reduction in time compared with existing approaches that employ semi-automatic or manual generation of HDL models.

Our future work will focus on generating HDL models for real world architectures. We have not considered the optimization and resource sharing issues of our data path components yet. As a result, the execution units consumes 50-60% of the total area and power of the generated hardware model. Our future research includes an improved methodology to generate optimized data path components with shared resources.

10 Acknowledgments

This work was partially supported by NSF grants CCR-0203813 and CCR-0205712. We would like to acknowledge the members of the ACES team for their inputs.

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] A. Hoffmann et al. A Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using the Machine Description Language LISA. In *ICCAD*, 2001.

[3] A Superscalar Version of the DLX Processor. *http://www.rs.e-technik.tu-darmstadt.de/TUD/res/dlxdocu/SuperscalarDLX.html*.

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

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

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

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

[8] Improv Inc. *http://www.improvsys.com*.

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

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

[11] M. Itoh and Y. Takeuchi and M. Imai and A. Shiomi. Synthesizable HDL Generation for Pipelined Processors from a Micro-Operation Description. *IEICE Trans. Fundamentals*, E00-A(3), March 2000.

[12] M. Itoh et al. PEAS-III: An ASIP Design Environment. In *ICCD*, 2000.

[13] O. Schliebusch et al. Architecture Implementation using the Machine Description Language LISA. In *VLSI Design / ASPDAC*, 2002.

[14] O. Wahlen et al. Application Specific Compiler/Architecture Codesign: A Case Study. In *LCTES-SCOPES*, 2002.

[15] 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.

[16] P. Mishra and J. Astrom and N. Dutt and A. Nicolau. Functional Abstraction of Programmable Embedded Systems. Technical Report UCI-ICS 01-04, University of California, Irvine, January 2001.

[17] P. Mishra and N. Dutt. Automatic Functional Test Program Generation for Pipelined Processors using Model Checking. In *HLDVT*, 2002.

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

[19] P. Mishra et al. Functional Abstraction driven Design Space Exploration of Heterogeneous Programmable Architectures. In *ISSS*, 2001.

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

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

[22] S. Aditya and B. R. Rau and V. Kathail. Automatic Architectural Synthesis of VLIW and EPIC Processors. In *ISSS*, 1999.

[23] Synopsys. *http://www.synopsys.com*.

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

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

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

[27] The SH-3 DSP RISC Embedded Processor. *http://www.hitachi.com*.

[28] The ST100 DSP-MCU Architecture. *http://www.st.com*.

[29] 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.