# Synthesis-driven Exploration of Pipelined Embedded Processors*

Prabhat Mishra
pmishra@cecs.uci.edu

Arun Kejariwal
arun_kejariwal@ieee.org

Nikil Dutt
dutt@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 advances on language based software toolkit generation enables performance driven exploration of embedded systems by exploiting the application behavior. There is a need for an automatic generation of hardware to determine the required silicon area, clock frequency, and power consumption of the candidate architectures. In this paper, we present a language based exploration framework that automatically generates synthesizable RTL models for pipelined processors. Our framework allows varied micro-architectural modifications, such as, addition of pipeline stages, pipeline paths, opcodes and new functional units. The generated RTL is synthesized to determine the area, power, and clock frequency of the modified architectures. 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.*

## 1 Introduction

The increasing complexity of modern embedded systems demands the use of rapid prototyping methodologies to provide an early estimation of the system area, power and performance. Recent approaches on Architecture Description Language (ADL) based software toolkit generation enables performance driven exploration, as shown in Figure 1. 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. However, the required silicon area, clock frequency, and power consumption can only be determined in conjunction with a synthesizable HDL model.

The exploration space for pipelined processors is enormous. System architects perform many micro-architectural explorations such as, addition/deletion of functional units, feedback paths, pipeline stages, pipeline paths, and instruction-set exploration prior to deciding an architecture for a given set of applications. Many commercial processors now offer the possibility of extending their instruction set for a specific application by introducing new instructions and customized functional units [1]. The goal of such processor extensions is typically to optimize performance in an application domain without violating the area and energy constraints. Algorithms have been proposed to decide automatically, from high level application code, which operations are to be carried out in the customized extensions [2]. These algorithms primarily use performance improvement as a metric for selecting new operations. However, area, power, and clock frequency should also be considered during exploration. It is necessary to generate synthesizable register-transfer level (RTL) models from the high level specification of the processor to enable area, power and performance driven instruction-set exploration.

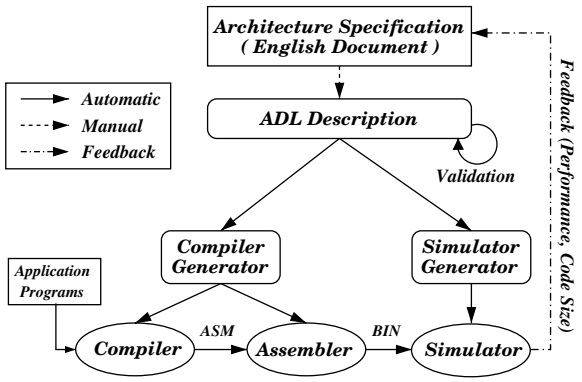


**Figure 1. ADL driven Design Space Exploration**

Manual or semi-automatic generation of synthesizable hardware description language (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, such as 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 a unified development environment, the design process is prone to error and may lead

*This work was partially supported by NSF grants CCR-0203813 and CCR-0205712.

to inconsistencies between hardware and software representations. Automatic generation of synthesizable HDL design along with a software toolkit from a single specification language will be an effective solution for early architectural exploration.

The contribution of this paper is a methodology for automatic generation of synthesizable hardware from a specification language to enable rapid exploration of pipelined processors. We use the EXPRESSION ADL [3] to specify the architecture. EXPRESSION has been used for generation of compiler and simulator. In this paper, we focus on synthesizable HDL generation from the ADL specification. 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 presents related work addressing language driven HDL generation approaches. Section 3 presents our ADL driven exploration framework using synthesizable HDL generation. Section 4 presents exploration experiments using our framework. Finally, Section 5 concludes the paper.

## 2 Related Work

There are two major approaches in the literature for synthesizable HDL generation. The first one is a parameterized processor core based approach. These cores are bound to a single processor template whose architecture and tools can be modified to a certain degree. The second approach is based on processor specification languages.

Examples of processor template based approaches are Xtensa [1], Jazz [4], and PEAS [5]. Xtensa [1] 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 [4] 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, depth of hardware task queue, and addition of custom functionality in Verilog. PEAS [5] 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.

Processor description language driven HDL generation frameworks can be divided into three categories based on the type of information the languages can capture. The languages such as nML [6] and ISDL [7] capture the instruction set (behavior) of the processor. On the other hand, the structure-centric languages such as MIMOLA [8] captures the net-list of the target processor. Finally, recent languages such as LISA [9] and EXPRESSION [3] captures both structure and behavior of the processor.

In nML, the processor instruction-set is described as an attributed grammar with the derivations reflecting the set of legal instructions. 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 on the generated HDL model. Itoh et al. [10] have proposed a micro-operation description based synthesizable HDL generation. It can handle simple processor models with no hardware interlock mechanism or multi-cycle operations. MIMOLA [8] 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.

LISA [9] captures operation-level description of the pipeline. The synthesizable HDL generation approach based on LISA language [11] 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 generates an 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 is the 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 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.

## 3 Architecture Exploration Framework

Figure 2 shows our ADL driven architecture exploration framework. 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 [12, 13]. The software toolkit, including compiler and simulator, is generated automatically from the ADL specification. The application program is compiled and simulated to generate performance numbers. The hardware implementation needs to be generated from the ADL specification to enable area, power and performance driven exploration.

In this paper, we present automatic generation of synthesizable HDL models from the ADL specification using functional abstraction. The functional abstraction technique was first introduced by Mishra et al. [14] for generating simulation models for a wide variety of architectures. First, we briefly describe the EXPRESSION ADL followed by a brief

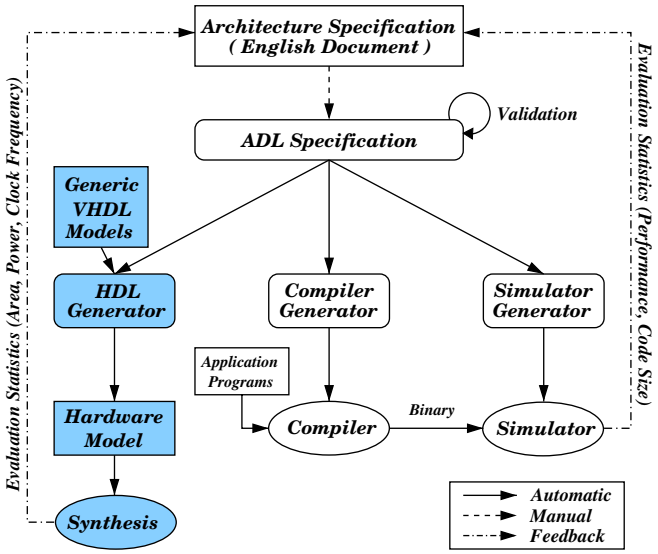


**Figure 2. Architecture Exploration Framework**

description of the functional abstraction technique. Finally, we describe the generation of HDL models. 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 [18] to generate evaluation statistics such as, area, clock frequency and power consumption.

### 3.1 The ADL Specification

The EXPRESSION ADL [3] captures the structure, behavior and mapping (between structure and behavior) of the processor. The structure contains the description of each component and the connectivity between the components. There are four types of components: *units* (e.g., ALUs), *storages* (e.g., register files), *ports*, and *connections* (e.g., buses). Each component has a list of attributes. For example, a functional unit will have information regarding latches, ports, connections, supported opcodes, execution timing, and capacity. The behavior is captured by describing the operations. Each operation is described in terms of it's opcode, operands, behavior and instruction format. Finally, the mapping functions map operations in the behavior to components in the structure. It defines, for each functional unit, the set of operations supported by that unit (and vice versa). For example, an operation *add* is mapped to *ALU* unit in a typical processor.

### 3.2 The Functional Abstraction

The functional abstraction technique was first introduced by Mishra et al. [14] for generating simulation models from the ADL specification. 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) may be used in different functional unit, or may have 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). They defined the necessary generic functions, sub-functions and computational environment needed to capture a wide variety of processor and memory features.

```
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 3. A Fetch Unit Example**

The structure of each functional unit is captured using parameterized functions. For example, a fetch unit functionality contains several parameters, such as number of operations read per cycle, number of operations written per cycle, reservation station size, branch prediction scheme, number of read ports, number of write ports etc. Figure 3 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-functions allows the flexibility of specifying the system in finer detail. It also allows reuse of the components.

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. For example, some common sub-functions are ADD, SUB, MUL, 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 4.

Similarly, they defined generic functions and sub-functions for memory modules, controller, interrupts, exceptions, DMA, and co-processor. The detailed description of generic abstractions for all of the micro-architectural components can be found in [14].

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

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

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

**Figure 4. Modeling of MAC operation**

### 3.3 Synthesizable HDL Generation

We have implemented all the generic functions and sub-functions using HDL. Our framework generates HDL description from the ADL specification of the processor by composing functional abstraction primitives. In this section, we briefly describe how to generate three major components of the processor: instruction decoder, datapath and controller, using the generic HDL models. The detailed description is available in [16].

**Generation of Instruction Decoder**

A generic instruction decoder uses information regarding individual instruction format and opcode mapping for each functional unit to decode a given instruction. The instruction format information is available in the operation description. The decoder extracts information regarding opcode, operands etc. from input instruction using the instruction format. The mapping section of the EXPRESSION ADL has 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 to send the instruction.

**Data Path Generation**

The implementation of datapath consists of two parts. First, compose each component in the structure. Second, instantiate components (e.g., fetch, decode, ALU, LdSt, write-back, branch, caches, register files, memories etc.) and establish connectivity using appropriate number of 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, appropriate number of operand read functionalities need to be instantiated unless the same read functionality can be shared using multiplexors. Similarly, if this execution 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 execution unit might contain many more functionalities e.g., read latch, write latch, and insert/delete/modify reservation station (if applicable).

**Generation of 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 the input instruction. For example, the local controller in an execution 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.

## 4 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 performing rapid exploration of the DLX architecture [15].

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 can be used for comparison purposes. Second, it has many interesting features, such as fragmented pipelines and multi-cycle functional units that are representative of many commercial pipelined processors such as TI C6x, PowerPC, and MIPS R10K.

### 4.1 Experimental Setup

The DLX architecture has five pipeline stages: fetch, decode, execute, memory, and writeback. The *execute* stage has four parallel paths: integer ALU (*IALU*), 7 stage multiplier (*MUL1 - MUL7*), four stage floating-point adder (*FADD1 - FADD4*), and a multi-cycle divider (*DIV*).

The EXPRESSION ADL captures the structure and behavior of the DLX architecture. Synthesizable HDL models are generated automatically from the ADL specification. We have used Synopsys Design Compiler [18] to synthesize the generated HDL description using LSI 10K technology libraries and obtained area, power and clock frequency values.

**Table 1. Synthesis Results: RISC-DLX vs Public-DLX**

| | HDL Code (lines) | Area (gates) | Speed (MHz) | Power (mW) |
|---|---|---|---|---|
| *RISC-DLX* | 7758 | 208 K | 35 | 32.6 |
| *Public-DLX* | 6529 | 159 K | 44 | 27.4 |

To ensure the functional correctness, the generated HDL model is validated against the generated simulator using Livermoore loops (LL1 - LL24) and multimedia kernels (com-

press, GSR, laplace, linear, lowpass, SOR and wavelet). To ensure the fidelity of the generated area, power, and performance numbers, we have compared our generated HDL (RISC version of the DLX) with the hand-written HDL model publicly available from *eda.org* [19]. Table 4.1 presents the comparative results between the generated DLX model (say *RISC-DLX*) and the hand written DLX model (say *Public-DLX*). Our generated design (*RISC-DLX*) is 20-30% off in terms of area, power and clock speed. We believe these are reasonable ranges for early rapid system prototyping and exploration.

```
arg = th2 * piovn
c1 = cos(arg)
s1 = sin(arg)
c2 = c1 * c1 - s1 * s1;
s2 = c1 * s1 + c1 * s1;
c3 = c1 * c2 - s1 * s2;
s3 = c2 * s1 + s2 * c1;
int4 = in * 4;
j0 = jr * int4 + 1;
k0 = ji * int4 + 1;
jlast = j0 + in - 1;
```

**Figure 5. The Application Program**

Figure 5 shows one of the most frequently executed code segment from FFT benchmark that we have used as an application program during micro-architectural exploration.

### 4.2 Results

We have performed extensive architectural explorations by varying different micro-architectural features [17]. In this section we present three exploration experiments: pipeline path exploration, pipeline stage exploration and instruction-set exploration. The reported area, power, and clock frequency numbers are for the execution units only. The numbers do not include the contributions from others components such as *Fetch*, *Decode*, *MEM* and *WriteBack*.

#### Addition of Functional Units (Pipeline Paths)

Figure 6 shows the exploration results due to addition of pipeline paths using the application program shown in Figure 5. The first configuration has only one pipeline path consisting of *Fetch*, *Decode*, one execution unit (say *Ex1*), *MEM* and *WriteBack*. The *Ex1* unit supports five operations: *sin*, *cos*, +, - and $\times$. The second configuration is exactly same as the first configuration except it has one more execution unit (say *Ex2*) parallel to *Ex1*. The *Ex2* unit supports three operations: +, - and $\times$. Similarly, the third configuration has three parallel execution units: *Ex1* (+, -, $\times$), *Ex2* (+, -, $\times$) and *Ex3* (*sin*, *cos*, +, - and $\times$). Finally, the fourth configuration has four parallel execution units: *Ex1* (sin, cos), *Ex2* (+, -, MAC[1]), *Ex3* and *Ex4*, where *Ex3* and *Ex4* are customized functional units that perform $a \times b + c \times d$.

The application program requires fewer number of cycles (schedule length) due to the addition of pipeline paths whereas the area and power requirement increases. The fourth configuration is interesting since both area and schedule length decrease due to addition of specialized hardware and removal of operations from other execution units.

[1]MAC performs multiply-and-accumulate of the form $a \times b + c$

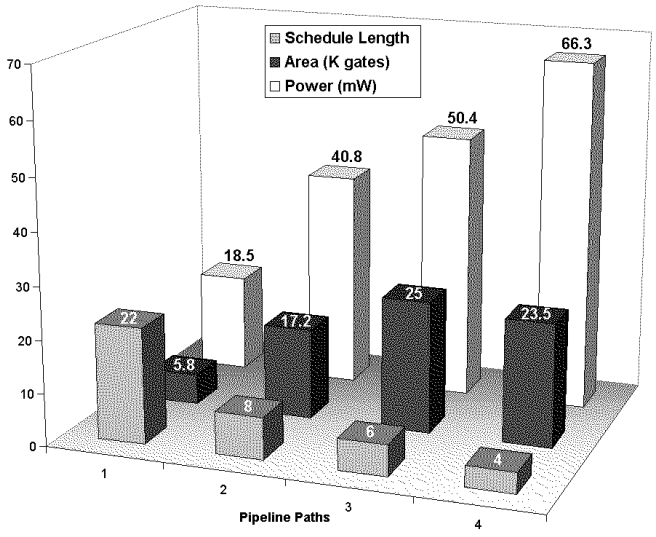


**Figure 6. Pipeline Path Exploration**

#### Addition of Pipeline Stages

Figure 7 presents exploration experiments due to addition of pipeline stages in the multiplier unit. The first configuration is a one-stage multi-cycle multiplier. The second, third and fourth configurations use multipliers with two, three and four stages respectively.

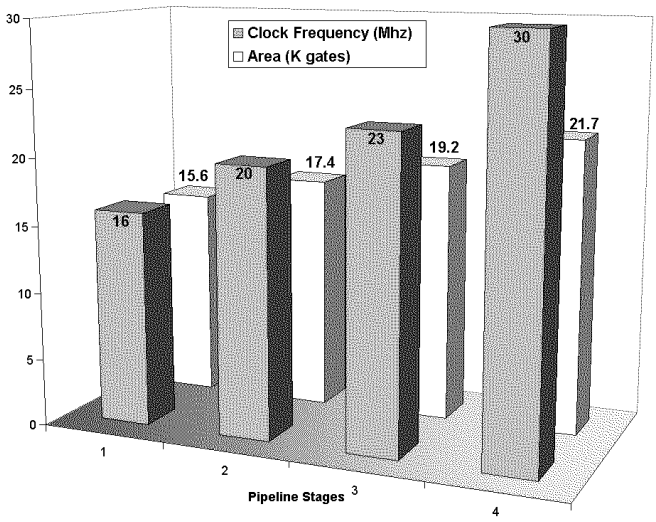


**Figure 7. Pipeline Stage Exploration**

The clock frequency (speed) is improved due to addition of pipeline stages. The fourth configuration generated 30% speed improvement at the cost of 13% area increase over the third configuration.

#### Addition of Operations

Figure 8 presents exploration results for addition of opcodes using three processor configurations. The three configurations are shown in Figure 8. The first configuration has four parallel execution units: *FU1*, *FU2*, *FU3* and *FU4*. The *FU1* supports three operations: +, -, and $\times$. The *FU2*, *FU3*

and *FU4* supports (+, -, ×), (*and*, *or*) and (*sin*, *cos*) respectively. The second configuration is obtained by adding a *cos* operation in the *FU3* of the first configuration. This generated reduction of schedule length of the application program at the cost of increase in area. The third configuration is obtained by adding multipliers both in *FU3* and *FU4* of the second configuration. This generated best possible (using +, -, ×, *sin* and *cos*) schedule length for the application program shown in Figure 5.

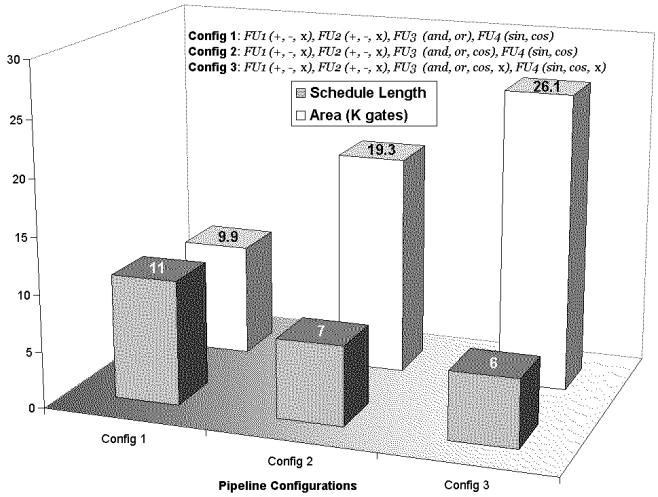


**Figure 8. Instruction-Set Exploration**

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 exploration time is at least an order of magnitude.

## 5 Summary

We have presented an ADL driven hardware generation and exploration framework for pipelined processors. 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 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 results show that a wide variety 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.

## References


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

[2] K. Atasu, L. Pozzi and P. Ienne. Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints. *Design Automation Conference (DAC)*, 2003.

[3] A. Halambi et al. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. *Design Automation and Test in Europe (DATE)*, 1999.

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

[5] M. Itoh et al. PEAS-III: An ASIP Design Environment. *International Conference on Computer Design (ICCD)*, 2000.

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

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

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

[9] V. Zivojnovic et al. LISA - Machine Description Language and Generic Machine Model for HW/SW Co-Design. *VLSI Signal Processing*, 1996.

[10] M. Itoh et al. Synthesizable HDL Generation for Pipelined Processors from a Micro-Operation Description. *IEICE Trans. Fundamentals*, E00-A(3), March 2000.

[11] O. Schliebusch et al. Architecture Implementation using the Machine Description Language LISA. *VLSI Design / ASP-DAC*, 2002.

[12] P. Mishra et al. Automatic Verification of In-Order Execution in Microprocessors with Fragmented Pipelines and Multicycle Functional Units. *DATE*, 2002.

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

[14] P. Mishra et al. Functional Abstraction driven Design Space Exploration of Heterogeneous Programmable Architectures. *International Symposium on System Synthesis (ISSS)*, 2001.

[15] J. Hennessy and D. Patterson. *Computer Architecture: A quantitative approach*. Morgan Kaufmann Publishers Inc.

[16] P. Mishra et al. Rapid Exploration of Pipelined Processors through Automatic Generation of Synthesizable RTL Models. *Rapid System Prototyping (RSP)*, 2003.

[17] A. Kejariwal et al. HDLGen: Automatic HDL Generation driven by an Architecture Description Language. Technical Report CECS 03-04, University of California, Irvine.

[18] Synopsys Design Compiler. *http://www.synopsys.com*.

[19] http://www.eda.org/rassp/vhdl/models/processor.html. *Synthesizable DLX: Generic 32-bit RISC Processor*.