EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. *

Ashok Halambi
ahalambi@ics.uci.edu

Peter Grun
pgrun@ics.uci.edu

Vijay Ganesh
ganesh@ics.uci.edu

Asheesh Khare
akhare@ics.uci.edu

Nikil Dutt
dutt@ics.uci.edu

Alex Nicolau
nicolau@ics.uci.edu

Department of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA

Abstract

We describe **EXPRESSION***, a language supporting architectural design space exploration for embedded Systems-on-Chip (SOC) and automatic generation of a retargetable compiler/simulator toolkit. Key features of our language-driven design methodology include: a mixed behavioral/structural representation supporting a natural specification of the architecture; explicit specification of the memory subsystem allowing novel memory organizations and hierarchies; clean syntax and ease of modification supporting architectural exploration; a single specification supporting consistency and completeness checking of the architecture; and efficient specification of architectural resource constraints allowing extraction of detailed reservation tables for compiler scheduling. We illustrate key features of EXPRESSION through simple examples and demonstrate its efficacy in supporting exploration and automatic software toolkit generation for an embedded SOC codesign flow.*

1 Introduction

The advent of System-on-Chip (SOC) technology has resulted in a paradigm shift for the design process of embedded systems employing programmable processors with custom hardware. Modern system-level design libraries frequently consist of Intellectual Property (IP) blocks such as processor cores that span a spectrum of architectural styles, ranging from traditional DSPs and superscalar RISC, to VLIWs and hybrid ASIPs. Furthermore, SOC technologies permit the incorporation of novel on-chip memory organizations (including the use of on-chip DRAM, frame buffers, streaming buffers, and partitioned register files), allowing a wide range of memory organizations and hierarchies to be explored and customized for the specific embedded application.

The embedded SOC designer is thus faced with the dual tasks of 1) rapidly exploring and evaluating different architectural and memory configurations, and 2) using a cycle-accurate simulator and retargetable optimizing compiler to adapt the application and architecture with the goal of meeting system-level performance, power and cost objectives. Furthermore, shrinking time-to-market cycles create an urgent need to perform the traditionally sequential tasks of hardware and software design in parallel. An effective embedded SOC codesign flow must therefore support automatic software toolkit generation, without loss of optimizing efficiency. This has resulted in a paradigm shift towards a *language-based design methodology* for embedded SOC optimization and exploration. Consequently there is tremendous interest in using Architectural Description Languages (ADLs) to drive design space exploration and automatic compiler/simulator toolkit generation.

As with an HDL-based ASIC design flow, several benefits accrue from a language-based design methodology for embedded SOC exploration, including the ability to perform (formal) verification and consistency checking, to modify easily the target architecture and memory organization for design space exploration, and to drive automatically the backend toolkit generation from a single specification. In this paper we describe **EXPRESSION**, an ADL that effectively supports these dual goals of SOC exploration, as well as automatic generation of a high-quality software toolkit for embedded SOC. Section 2 describes our goals and approach. In Section 3, we describe related work on ADLs and compare them with EXPRESSION. Sections 4 and 5 present a brief overview of EXPRESSION using an example architecture. Section 6 describes EXPRESSION's support for detailed scheduling and resource constraint specification. Section 7 illustrates the ease of modification in EXPRESSION to support design space exploration, while Section 8 concludes this paper.

2 Goals and Approach

SOC designers spend a lot of time and effort exploring candidate processor architectures. The availability of a variety of processor core IP libraries (including DSP, VLIW, SS/RISC and ASIP) presents the system designer with a large exploration space for the choice of a base processor architecture. Thus, tool-kits which allow the designer to perform rapid exploration of various processor alternatives are necessary. These tool-kits must provide the designer with quantitative performance measurements in order for him to perform intelligent tradeoffs. Furthermore, the stringent performance, power, code density, and

*This work was partially supported by grants from NSF (MIP-9708067) and ONR (N00014-93-1-1348).

cost constraints mandated by modern embedded systems necessitate the development of a high-quality software tool-kit, including, at a minimum, a cycle-accurate simulator, and an optimizing Instruction-Level Parallelism (ILP) compiler that can exploit novel memory organizations.

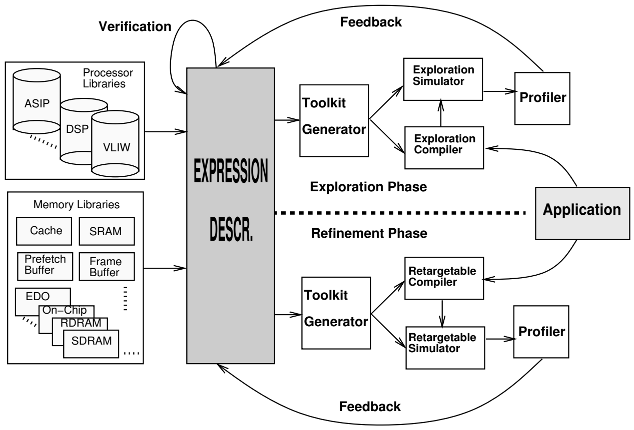

Figure 1. EXPRESSION Design Flow

The system designer also requires the ability to customize the base processor by changing parameters of the processor core (e.g. number of functional units, operation latencies). The memory-intensive nature of many embedded applications (e.g. multimedia and network) further exacerbates the traditionally critical memory bottleneck. This requires the ability to explore (and optimize for) novel on-chip and off-chip memory organizations and hierarchies to improve memory bandwidth (examples include the use of on-chip DRAM, frame-buffers, queues, novel cache hierarchies, etc.). An important aspect of such an exploration (not taken into account by most other approaches) is the ability to also customize the compiler concurrently with the processor such that a "best-fit" is obtained.

Figure 1 shows our language-based design methodology using EXPRESSION. An EXPRESSION description of an embedded SOC architecture can be used in two modes. In the *Exploration Phase*, the system designer explores and evaluates different base processor candidates (selected from the Processor Libraries), and different memory organizations and hierarchies (with components selected from the Memory Libraries). In the exploration phase, the toolkit generator is used to produce an *Exploration Simulator* and a *Exploration Compiler*. The goal here is to support rapid Design Space Exploration (DSE) with fast (possibly functional) simulation, and using the compiler in an estimation mode for comparative evaluation of candidate base processors and memory organizations. In the *Refinement Phase*, the EXPRESSION description is used to generate a cycle-accurate simulator and an optimizing ILP compiler that allows the system designer to tune the base processor characteristics, as well as to tune the memory subsystem hierarchy.

EXPRESSION was designed to provide a natural and easy-to-specify mechanism for capturing the information needed to support this ADL-based design space exploration and software toolkit generation methodology. As shown in Figure 1, EXPRESSION facilitates the automatic generation of an optimizing compiler and simulator. The retargetable compiler exploits the parallelism and pipelining available, while the simulator provides accurate timing and utilization information. Furthermore, since the description of complex processors is cumbersome and error-prone, EXPRESSION provides the ability to perform consistency checking and verification of the input specification.

3 Related Work

Traditionally, ADLs have been classified into two categories depending on whether they primarily capture the Instruction-Set (IS) or the structure of the processor.

nML [8] and **ISDL** [2] are examples of IS ADLs. In nML, the processor's IS is described as an attributed grammar with the derivations reflecting the set of legal instructions. nML has been used by the retargetable code generation environment CHESS [1] to describe DSP and ASIP processors. However, nML does not directly support multi-cycle or multi-word instructions. ISDL also describes the processor in terms of its IS, with the goal of deriving a code-generator ([12]), assembler and simulator. In ISDL, constraints on parallelism are explicitly specified through illegal operation groupings. This could be tedious for complex architectures like DSPs which permit operation parallelism (e.g. Motorola 56K) and VLIW machines with distributed register files (e.g. TI C6X). The retargetable compiler system by Yasuura et al.[3] 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 [13] is an example of an ADL which primarily captures the structure of the processor wherein the net-list of the target processor is described in a HDL like language. One advantage of this approach is that the same description is used for both processor synthesis, and code generation. The target processor has a micro-code architecture. The net-list description is used to extract the instruction set [6], and produce the code generator. Extracting the instruction set from structure may be difficult for complicated instructions, and may lead to poor quality code. MIMOLA descriptions are generally very low-level, and laborious to write. It is not clear how they generate a cycle-accurate simulator using the MIMOLA description.

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** [7] is one such ADL whose main characteristic is the operation-level description of the pipeline. LISA seems to have been designed primarily for retargeting simulators. Also, it does not support specification of detailed constraint information needed for compiler instruction scheduling. **RADL** [16] is an extension of the LISA approach that focuses on explicit support of detailed pipeline behavior to enable generation of production quality cycle- and phase-accurate simulators. **FLEXWARE** [5] and **MDes** [18] 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. However, it is not clear how the resource conflicts between parallel/pipelined instructions, needed in scheduling, are captured. Furthermore, explicit specification of the memory subsystem does not appear to be possible.

The **MDes** [11] language used in the Trimaran [18] system is closest in its goals to our approach. It is a mixed-level ADL, intended for DSE. Information is broken down into sections (such as format, resource-usage, latency, operation, register etc.), based on a high-level classification of the information being represented. MDes has good constructs for preprocessing which enable concise descriptions. 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, etc.). In our approach we target more general memory organizations and hierarchies, including on-chip DRAM, frame buffers, partitioned memory address spaces, etc. MDes captures constraints between operations with explicit Reservation Tables [4], using a hierarchical description for compactness. Because this hierarchical specification allows instruction set and structure information to be specified at any level of the hierarchy, (e.g. usage, latency, operation), local changes to the architecture (for exploration) could propagate through the hierarchy, and require global changes to the specification making it cumbersome and error-prone.

In EXPRESSION we also follow a mixed-level approach (behavioral and structural) to facilitate DSE. Furthermore, we provide support for specification of novel memory subsystems. We avoid explicit representation of the reservation tables by extracting them from the structural description (using the algorithm outlined in Section 6) Our approach also eliminates redundancy by using the same net-list information to drive both the compiler and the simulator.

4 EXPRESSION Overview

EXPRESSION captures enough information to retarget a cycle-accurate structural simulator and an optimizing ILP compiler. The system is described both in terms of its instruction-set (Behavior Specification) and its structure as shown in Figure 2. Section 5 illustrates these with the aid of an example.

Figure 2 shows the interaction between EXPRESSION specification and the retargetable compiler/simulator. The dark shaded boxes represent generators that read in the appropriate sections of the EXPRESSION specification and generate the information required by the compiler/simulator components. The structural simulator is retargeted by changing the datapath netlist and the execution semantics of the architectural components. The ILP compiler is retargeted by changing the machine dependent parameters of Instruction Selection, Resource Allocation, ILP Scheduling and Memory Optimization techniques.

A key feature required by an ADL supporting aggressive and accurate compiler scheduling is the ability to capture detailed resource constraint information, typically in the form of reservation tables. Previous approaches (e.g. MDes) required the user to specify reservation tables on a per-operation basis. This makes specification of reservation tables cumbersome for architectures containing a lot of instruction types (e.g. instructions with varied accessing modes). Furthermore, for VLIW architectures with DSP-style features, the IS may not be well-structured; thus one cannot use hierarchical composition rules to simplify the specification of reservation tables. In EXPRESSION we solve this problem by permiting concise specification of resource constraints at an abstract level, from which detailed reservation tables can be automatically generated on a per-operation basis.

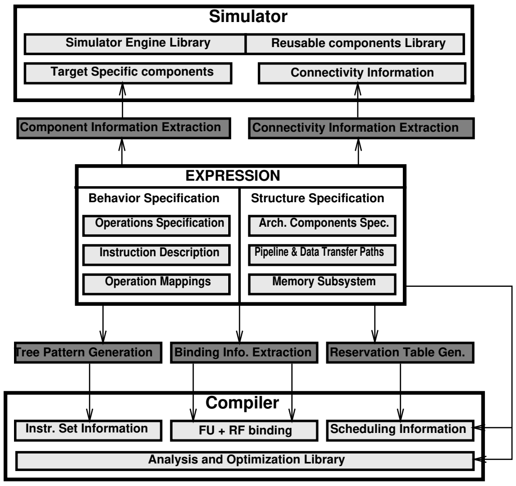

Figure 2. Interaction between EXPRESSION and the retargetable compiler/simulator toolkit.

EXPRESSION allows the user to specify the RT–level datapath netlist[1] (i.e. netlist without the control signals) of the processor at an abstract level. First, each RT-level architectural component is specified. Then, pipeline paths and all valid data-transfer paths are specified. Using this information, both the netlist for simulator and reservation tables for compiler are generated. Our resource constraint specification scheme not only reduces the complexity of specification, but also allows for consistency and completeness checking on the specification.

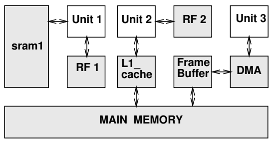

Figure 3. An example Memory Subsystem

Another key feature of EXPRESSION is the ability to specify novel memory subsystems. SOC technology permits customization of processor cores with different memory architectures and thus requires the exploration of novel memory organizations and hierarchies. Figure 3 shows an example memory system with the memory address space divided between a

[1] subsequently, netlist will refer to RT–level datapath netlist

SRAM and a main memory (DRAM). The main memory is accessed through the data cache and the frame buffer. EXPRESSION's Storage Subsytem specification is used to retarget compiler optimizations that exploit memory organization.

5 EXPRESSION Organization

EXPRESSION employs a simple LISP-like syntax to ease specification and enhance readability. A detailed description of EXPRESSION can be found in[10]. We illustrate salient features of EXPRESSION using the example in Figure 4.

5.1 Example Architecture

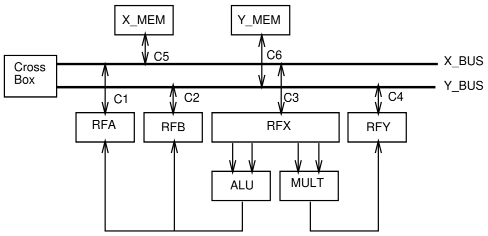

Figure 4. An example architecture

Figure 4 shows a simplified version of the Motorola DSP 56000 processor to which an additional multiplier unit has been added. The DSP 56000 is a bus based architecture and can execute one ALU operation and upto two additional parallel moves in one cycle. Two Address Generation Units AGU1 and AGU2 (not shown in the figure) generate the addresses required for the parallel moves.[2]

5.2 Sections in EXPRESSION

As shown in Figure 2, an EXPRESSION description is composed of two main sections: Behavior (or IS), and Structure, which are further sub-divided into three subsections each: Operations, Instruction, Operation mappings, and Components, Pipeline/Data–transfer paths, Memory subsystem respectively. Each subsection is illustrated with examples below. (refer Figure 4)

5.2.1 Operations Specification

This subsection describes the IS of the processor. Each operation of the processor is described in terms of its opcode and operands. The types and possible destinations of each operand are also specified.

```
// Specifying the parallel move operations
(OP_GROUP pmove1_ops
(OPCODE pmove1 (OPERANDS src1 dst1)))
// The var groups (src1, dst1) used above are defined here.
(VAR_GROUPS
(src1 (OR X_mem Y_mem)) (dst1 (OR RFA RFB RFX RFY)))
```

5.2.2 Instruction Description

This subsection captures the parallelism available in the architecture. 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.

[2] While the 56K does not contain explicit opcodes to model the parallel moves, for purposes of illustration, we denote moves by opcodes 'pmove1'and 'pmove2'.

```
// The instr has 4 slots (alu, mult and 2 parallel moves)
(INSTR (SLOTS (UNIT ALU) (UNIT MULT)
(UNIT AGU1) (UNIT AGU2)))
```

5.2.3 Operation Mappings

In this subsection the user specifies information needed by Instruction Selection and architecture-specific optimizations of the compiler.

```
// mapping a compiler operation to a target specific op.
(OP_MAPPING ((GENERIC (iadd src1 src2 dst))
(TARGET (add src1 src2 dst)))
// Multiply x by 2 can be replaced with ADD x x
((GENERIC (mult src1 #2 dst))(TARGET (add src1 src1 dst)))
```

5.2.4 Components Specification

This subsection describes each RT-level component in the architecture. The components can be any of Pipeline units, Functional units, Storage elements, Ports, Connections and Buses. For multi-cyle or pipelined units, the timing behavior is also specified.

```
(ExUnit ALU() (OPCODES alu_ops)) // Instantiate ALU.
// alu_ops is defined in the Operations section.
```

5.2.5 Pipeline and Data-Transfer Paths Description

This subsection describes the netlist of the processor. The *pipeline description* provides a mechanism to specify the units which comprise the pipeline stages, while the *data-transfer paths description* provides a mechanism for specifying the valid data-transfers. This information is used to both retarget the simulator, and to generate reservation tables needed by the scheduler, as shown in Section 6

```
(PIPELINE FETCH DECODE EX) // 3 stage pipeline
(EX (PARALLEL ALU Mult AGU1 AGU2)
// Describe datapath transfers. Type: uni/bi-directional.
// (Source, sink, components activated during transfer)
(DTPATHS (TYPE BI)
(RFA X_bus C1) (RFB Y_bus C2) (RFX X_bus C3)
(RFY Y_bus C4) (X_mem X_bus C5) (Y_mem Y_bus C6)
(X_bus Y_bus CrossBox))
```

5.2.6 Memory Subsystem

This subsection describes the types and attributes of various storage components (like Register Files, SRAMs, DRAMs, Caches etc). Note that the netlist also contains connectivity information for the memory subsystem.

```
(STORAGE_PARAMETERS
(L1_cache (TYPE cache) (SIZE 1024) (LINE 32)
(ASSOCIATIVITY 2) (ADDRESS_RANGE 0 511)
(ACCESS_TIMES 1)))
```

6 Reservation Table Generation

As mentioned before, the compiler's scheduler needs reservation tables to test for resource conflicts between operations whose execution cycles overlap. Rather than requiring the user to laboriously specify the reservation tables on a per–operation basis, we use the structural information in EXPRESSION to automatically generate them. The key idea behind the reservation table generation approach in EXPRESSION is that every operation proceeds through a pipeline path and accesses storage units through some data-transfer paths. In effect, it is possible to trace the execution of the operation through the pipeline and data-transfer segments and thus generate accurate reservation tables. This frees the user from the burden of having to specify reservation tables for each operation and additionally, makes it possible to check for consistency and completeness in the specification.

In Figure 5 we outline the reservation table generation scheme, given an operation and the structural description. We also illustrate the generation of the reservation table for a parallel move operation (opcode 'pmove1' and functional unit 'AGU1').

Steps needed for Reservation Table Generation.	Example to illustrate Reservation Table Generation.
Input: Operation (OP)	Input: *pmove1 (AGU1) X_mem[100] RFB*
Output: Reservation Table (RT) for operation OP.	Output: /* Reservation Table for the operation */ /* Cyc stands for Cycle */
1. Find pipeline path which contains OP's F.U.	Pipeline path - Fetch; Decode; AGU1;
2. Generate partial RT from the pipeline path.	Partial RT - Cyc1: Fetch; Cyc2: Decode; Cyc3: AGU1;
3. **For** each data-transfer initiated by OP	
Determine the source and sink components.	Source: X_mem, Sink: RFB
Find transfer segments which implement. the data-transfer.	(C5 X_bus) (Cross Box) (Y_bus C2)
Add objects in the segment to RT.	RT - Cyc1 ... Cyc3: AGU1, X_mem, C5, X_bus, Cross Box, Y_bus C2 RFB
/* Output the completed reservation table */	

Figure 5. The steps in Reservation Table generation illustrated with an example (also refer Figure 4.)

As shown in Figure 1, EXPRESSION is designed to support both rapid DSE (exploration phase requiring fast turnaround time), as well as high-quality code generation with cycle-accurate simulation (refinement phase). Thus, in different phases of Figure 1's design flow, the computation speed requirement shifts from total exploration time to compilation and simulation time. To account for these conflicting goals, Reservation Tables can either be generated on-the-fly during compilation, or they can be generated beforehand and stored in a database. During the exploration phase computing reservation tables on-the-fly is beneficial. Once the architecture has been fixed, and a quality tool-kit is needed, the reservation tables are computed apriori to reduce compilation time.

7 Support for Design Space Exploration

DSE allows the system designer to perform tradeoffs between various competing goals like cost, performance and power. The objective during DSE is to evaluate numerous processor and memory configurations w.r.t. the targeted applications. Thus, an ADL for DSE should capture both structural and behavioral aspects of the system. Hence, EXPRESSION follows a mixed-level approach (to enable changes to structure or IS or the memory subsystem). Furthermore, the ADL should be able to easily reflect the changes made to the system. For example, the designer may vary the processor parameters like number of functional units, register files, buses etc. Another example could be varying the IS by adding/deleting operations, changing the operand types etc. Changes to the behavior are fairly easy in EXPRESSION as well as in other ADLs that are behavior-centric (e.g. ISDL) or which employ a mixed-level approach (e.g. MDes). On the other hand, structural changes in EXPRESSION are quite simple while structural changes in other ADLs (e.g. ISDL, MDes) become complicated, and in some cases (e.g. for novel memory organization and hierarchies) they are not feasible.

We evaluated EXPRESSION to see how easy it is to make local changes to the architecture and to add/delete operations and compared this with equivalent changes in MDes. We illustrate this on the example architecture shown in Figure 4. For this example, the initial MDes specification required approximately 100 lines while EXPRESSION required approximately 75 lines.[3]

Although the MDes specification appears to be fairly concise, this comes at a large cost: modifications to the architecture in MDes may necessitate a change in many sections. This increases the possibility of errors creeping into the specification. In EXPRESSION, changes to the architecture usually involve making changes only to certain sections. Furthermore, we can automatically generate detailed reservation table information from EXPRESSION's structural description, whereas MDes requires a per-operation change in the affected reservation tables. Additionally, the consistency and completeness checking mechanism alerts the designer to errors (if any) in the EXPRESSION specification.

We evaluate the effect of splitting the register file RFX in Figure 4 into two register files RFX1 and RFX2. (This could be useful, for instance, to reduce the cost since it is expensive to build register files with many ports). The modified architecture is shown in Figure 6.

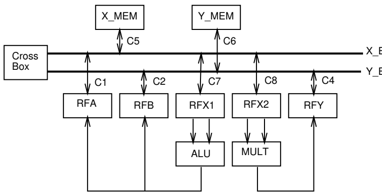

Figure 6. The modified example with RFX split into RFX1 & RFX2, connection C3 deleted, C7 & C8 added

This change in the architecture impacts the resource constraints specification of both MDes and EXPRESSION. Shown below are the additions[4] made to the specifications of both MDes and EXPRESSION in order to incorporate the changes to the example architecture (Figure 6).

7.1 Changes to MDes:

```
SECTION Resource{ RFX1(); RFX2(); C7(); C8(); }
SECTION Resource_Usage{
$for ( C7 C8 RFX1 RFX2){
RU_${CASE} (use(${CASE}) time(0));}}
SECTION Resource_Units{
RFX1_unit (use(RFX1 C7 X_bus));
RFX2_unit (use(RFX2 C8 X_bus));}
SECTION Table_Option{
any_X_reg_transfer (one_of(RFX1_unit RFX2_unit));}
SECTION Reservation_Table{
RT_PMOVE2_X2 (use(RU_AGU2 RFA_unit RU_RFX2));
RT_PMOVE2_X1_X2 (use(RU_AGU2 RFX1_unit RU_RFX2));}
SECTION Scheduling_Alternative{
```

[3] Due to space limitations, the complete specfications for MDes and EXPRESSION were omitted. These can be viewed at http://www.ics.uci.edu/˜ahalambi/EXPRESS/ADL/main.html

[4] The deletions made to remove the original register file were slightly more extensive in MDes than in EXPRESSION.

```
$for (CASE in PMOVE2_X2 PMOVE2_X1_X2){
ALT_${CASE} (resv(RT_${CASE}));}}
SECTION Operation{
$for (CASE in PMOVE2_X2 PMOVE2_X1_X2){
OP_${CASE} (alt(ALT_${CASE}));}}
```

7.2 Changes to EXPRESSION:

```
(RegFile RFX2()) (Connection C7() C8())
(DTPATHS (TYPE BI
(RFX1 X_bus C7) (RFX2 X_bus C8)))
```

For this local architectural change, MDes required modifications across 7 sections in the hierarchy (18 lines added), whereas in EXPRESSION the changes were more localized, requiring modifications to 2 sections (3 lines added). More complex changes to the architecture (such as adding buses, connection, units and memories) would require drastic changes in MDes, but would be rather simple in EXPRESSION. Thus, as can be seen from the above example, EXPRESSION seems to be better suited as a language for rapid DSE.

8 Summary

In this paper we present EXPRESSION, a new ADL used for rapid DSE, and retargeting a high-quality compiler/simulator toolkit. To support fast iteration through the design cycles of complex Systems-on-Chip, fast retargeting of the compiler and simulator is necessary. We use the EXPRESSION language to retarget the optimizing compiler and cycle-accurate simulator, through a set of toolkit generation algorithms. The compiler exploits the parallelism/pipelining available in the architecture, while the simulator provides accurate profiling feedback to the designer, to allow good system-level trade-offs.

Attributes	nML	LISA	ISDL	MIMOLA	MDes	EXPRESSION
Compiler supp	√	-	√	√	√	√
Cycle-acc sim	-	√	?	?	-	√
Pipeline supp	?	√	-	√	√	√
Mult-cyc supp	-	√	√	√	√	√
Net-list info	-	-	-	√	√	√
Instr-set info	√	√	√	-	√	√
Reserv tables	-	√	-	√	√	√
Memory hier	-	-	-	-	?	√

Table 1. Comparison between different ADLs

We also propose a new mechanism to concisely specify resource information required to generate reservation tables used in scheduling parallel/pipeline/multi-cycle operations from a structural description of the architecture. This eliminates redundancy, reducing the amount of description needed from the user, and makes the process less error-prone.

The memory-intensive nature of many multimedia applications requires exploration of different memory organizations in order to meet the cost/performance/power goals. EXPRESSION is able to capture these memory organizations for memory-aware compiler optimizations.

Table 1 summarizes the comparison between EXPRESSION and some other ADLs. The rows represent characteristics of the ADLs. In many respects the MDes language is similar to our approach. However, for cycle-accurate simulation, MDes provides a limited retargeting for the HPL-PD processor family, while we employ an approach similar to LISA, providing more control over the modeled architecture. We differ from LISA in the extensive support for optimizing compiler algorithms. Unlike MDes, we extract reservation tables from the netlist. Also, unlike other ADLs, EXPRESSION is able to capture novel memory subsystems. In EXPRESSION we employ a mixed-level specification combining both behavioral and structural information to efficiently support DSE and retargetability through automatic compiler/simulator toolkit generation.

EXPRESSION descriptions for a range of architectures including VLIW (TI C6X [17]) and DSP (Motorola 56K [14]) have been implemented. EXPRESSION is currently used to drive several projects at U.C. Irvine, including MEMOREX (a memory estimation and exploration environment)[9],[15] and EXPRESS (an optimizing, retargetable compiler and exploration environment for Embedded Systems-on-Chip). Our on-going work targets strengthening the coupling between the compiler and the architecture, by inserting more architecture–dependent optimizations and providing an architecture based self–adapting compiler flow.

References

[1] G. Goosens et al. CHESS: Retargetable code generation for embedded DSP processors. In *Code Generation for Embedded Processors.* Kluwer, 1997.

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

[3] H. Yasuura et al. A programming language for processor based embedded systems. In *Proc. APCHDL*, 1998.

[4] J. Gyllenhaal et al. Optimization of machine descriptions for efficient use. In *Proc. 29th Annual International Symposium on Microarchitecture*, 1996.

[5] P. Paulin et al. FlexWare: A flexible firmware development environment for embedded systems. In *Proc. Dagstuhl Code Generation Workshop*, 1994.

[6] R. Leupers et al. Retargetable generation of code selectors from HDL processor models. In *Proc. EDTC*, 1997.

[7] 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.

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

[9] P. Grun, F. Balasa, and N. Dutt. Memory size estimation for multimedia applications. In *Proc. CODES/CACHE*, 1998.

[10] P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt, and A. Nicolau. EXPRESSION: An ADL for system level design exploration. Technical Report TR 98-29, University Of California, Irvine, 1998.

[11] J. Gyllenhaal. A machine description language for compilation. Master's thesis, Dept. of EE, UIUC,IL., 1994.

[12] S. Hanono and S. Devadas. Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator. In *Proc. DAC*, 1998.

[13] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1), 1998.

[14] MOTOROLA Inc. *DSP56000 24-Bit Digital Signal Processor Family Manual*, 1995.

[15] P. Panda, N. Dutt, and A. Nicolau. Architectural exploration and optimization of local memory in embedded systems. In *Proc. ISSS*, 1997.

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

[17] TEXAS INSTRUMENTS. *TMS320C62x/C67x CPU and Instruction Set Reference Guide*, 1998.

[18] Trimaran Release: http://www.trimaran.org. *The MDES User Manual*, 1998.