EXPRESSION: An ADL for System Level Design Exploration

Peter Grun
(pgrun@ics.uci.edu)
Ashok Halambi
(ahalambi@ics.uci.edu)
Asheesh Khare
(akhare@ics.uci.edu)
Vijay Ganesh
(ganesh@ics.uci.edu)
Nikil Dutt
(dutt@ics.uci.edu)
Alexandru Nicolau
(nicolau@ics.uci.edu)

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

Technical Report #98-29
September 1998

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 modifications 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.

Contents

List of Figures

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 modifications 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 [17].

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 modifications to 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, stringent performance, power, code density, and 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 (a good description of memory-aware compiler techniques can be found in [17]).

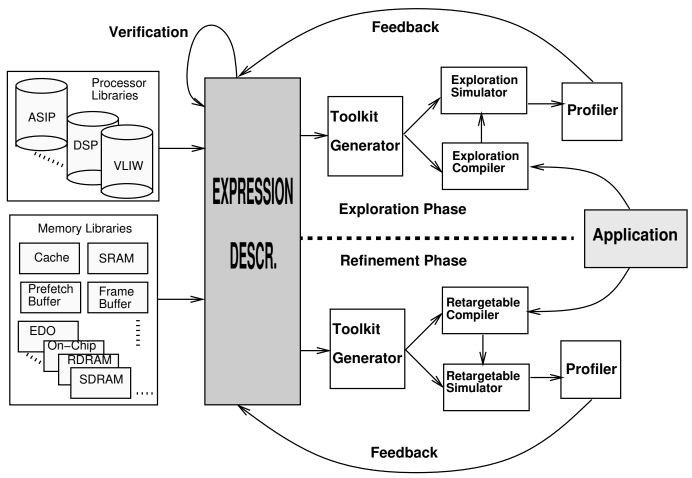

Figure 1: **EXPRESSION** Design Flow

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, requiring 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 (which is 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 [4] and **ISDL** [8] 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 [13] 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 ([9]), 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.[10] 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 [14] 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 [2], 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** [21] 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** [19] 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 [18] and **MDes** [1] 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** [6] language used in the Trimaran [1] 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 [7], 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.

The previous systems specify the parallelism conflicts between operations either by explicitly enumerating all *legal* combinations of instructions (nML), enumerating all *illegal* combinations of instructions (ISDL), or by explicit specification of the reservation tables (LISA, MDes). In either case, the user has to explicitly specify this information. Even though in MDes they optimized the representation of the reservation tables ([7]) by extracting the commonality inherent in the reservation tables (which both reduces the size of the reservation tables specification and increases the speed of scheduling), the user still needs to manually write the detailed reservation tables in a separate section of the description, which can be tedious. Moreover, the hierarchy introduced to minimize the description makes structural changes to the architecture (e.g., adding/deleting busses, connections, functional units etc.) very non-intuitive, by requiring changes at many levels of the hierarchy. It is possible to alleviate this problem, as in our approach. There is a lot of commonality between the information needed to generate the reservation tables and the structural simulator. In effect it is possible to generate the reservation tables from structural information already present in the EXPRESSION specification, to be used by the simulator.

We avoid explicit representation of the parallelism conflicts by extracting the reservation tables from the structural description, using the algorithm detailed 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.

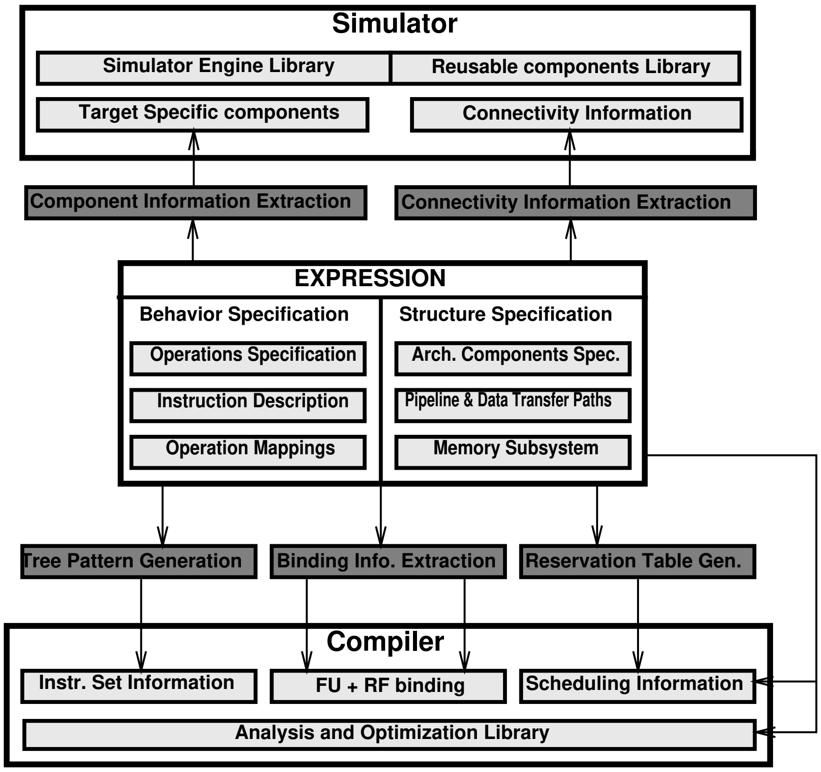

Figure 2: Interaction between EXPRESSION and the retargetable compiler/simulator toolkit.

Figure 2 shows the interaction between EXPRESSION specification and retargetable compiler and 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 net-list 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 permitting concise specification of resource constraints at an abstract level, from which detailed reservation tables can be automatically generated on a per-operation basis.

EXPRESSION allows the user to specify the RT–level datapath net-list[1] (i.e. net-list 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 net-list for simulator and reservation tables for compiler are generated. This contrasts to other approaches which require the user to specify the constraint information for operations as a separate section of the architecture specification (e.g. ISDL, MDes).

Our approach allows for a natural and unified description of the architecture from which the relevant information (net-list for structural simulator and reservation tables for compiler) is extracted. This scheme not only reduces the complexity of specification, but also allows for consistency and completeness checking on the specification as explained in Section 7.

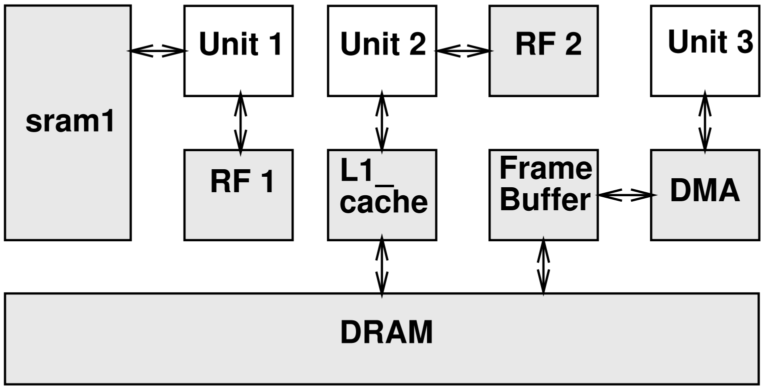

Figure 3: An example Memory Subsystem

Another key feature of EXPRESSION is the ability to specify novel memory subsystems. SOC technology permits customization of processor cores with different memory architectures [12] 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 SRAM and a DRAM. The DRAM is accessed through the data cache and the frame buffer. EXPRESSION's Storage Subsystem specification is used to retarget compiler optimizations that exploit such non-traditional memory organizations.

5 EXPRESSION Organization

EXPRESSION employs a simple LISP-like syntax to ease specification and enhance readability. Appendix A contains a detailed BNF description of the language. We illustrate salient features of EXPRESSION using the example in Figure 4.

5.1 Example Architecture

Figure 4 shows a driver example used to illustrate features of EXPRESSION. The architectures resembles the Motorola DSP 56000 [15] processor to which an additional multiplier unit has been added, along with a memory hierarchy composed of two caches and a SRAM. The DSP 56000 is a bus based architecture and can execute one ALU operation and up to 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

[1] subsequently, net-list will refer to RT–level datapath net-list

[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'.

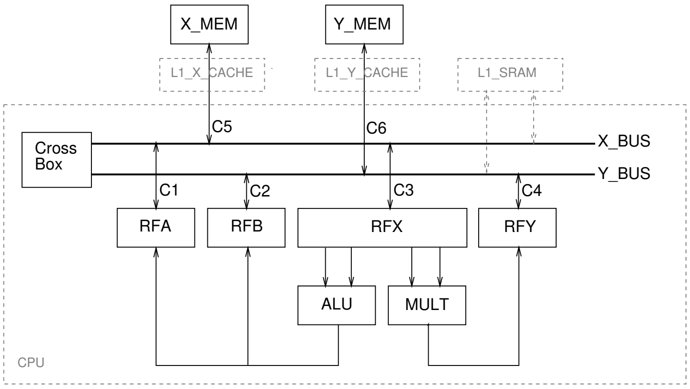

Figure 4: An example architecture

mappings, and Components, Pipeline/Data–transfer paths, Memory subsystem respectively. Each subsection is illustrated with examples below (using the example architecture in Figure 4).

5.2.1 Operations Specification

This subsection describes the IS of the processor. The IS is organized into operation groups, with each group containing a set of operations having some common characteristics. In the example architecture the operations are grouped into alu_ops, mult_ops, pmove1_ops and pmove2_ops. Each operation is then described in terms of its opcode, operands and behavior. Each operand is classified either as source or as destination. Further, each operand has an associated list of register files to which it can be bound. These lists are specified in the **VAR_GROUPS** subsection. For instance, the ADD operation gets its sources from RFX and writes to RFA and RFB. Thus, RFX forms one VAR_GROUP (g1), while RFA | RFB forms another (g2). Below, we give the specification for some representative operations (including the ADD operation).

```
(OP_GROUP alu_ops
    (OPCODE add
        (OP_TYPE DATA_OP)
        (OPERANDS (SRC1 g1) (SRC2 g1) (DST g2))
        (BEHAVIOR DST = SRC1 + SRC2)
    )
    (OPCODE sub
        (OP_TYPE DATA_OP)
        (OPERANDS (SRC1 g1) (SRC2 g1) (DST g2))
        (BEHAVIOR DST = SRC1 − SRC2)
    )
)
(OP_GROUP mult_ops
    (OPCODE mult
        (OP_TYPE DATA_OP)
        (OPERANDS (SRC1 g1) (SRC2 g1) (DST g3))
        (BEHAVIOR DST = SRC1 * SRC2)
    )
)
(OP_GROUP pmove1_ops
    (OPCODE pmove1
        (OP_TYPE DATA_OP)
        (OPERANDS (SRC1 g4) (DST g4))
        (BEHAVIOR DST = SRC1)
    )
)
(OP_GROUP pmove2_ops
    (OPCODE pmove2
        (OP_TYPE DATA_OP)
        (OPERANDS (SRC1 g2) (DST g5))
        (BEHAVIOR DST = SRC1)
    )
```

```
)
(VAR_GROUPS
	(g1 RFX)
	(g2 RFA | RFB)
	(g3 RFY)
	(g4 X_mem | Y_mem RFA | RFB | RFX | RFY)
	(g5 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.

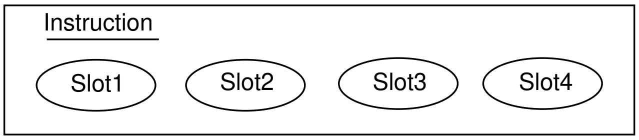

Figure 5: Illustration of the instruction for the example architecture

```
// The instr has 4 slots (alu, mult and 2 parallel moves)
(INSTR
	(WORDLEN 32)
	(SLOTS
		((TYPE DATA) (BITWIDTH 8) (UNIT ALU))
		((TYPE DATA) (BITWIDTH 8) (UNIT MULT))
		((TYPE DATA) (BITWIDTH 8) (UNIT AGU1))
		((TYPE DATA) (BITWIDTH 8) (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.

Each entry in this subsection represents the mapping of a (sequence of) operation to another (sequence of) operations. The mapping can be from generic compiler operations to target processor operations, in which case it is used by the instruction selection algorithm, or from target operations to target operations, to be used as architecture-dependent optimizations. The instruction selection algorithm uses a tree parsing technique utilizing dynamic programming. The current implementation uses the lburg [3] tree parser generator, which accepts as input a tree grammar. This tree grammar is generated by transforming the operation sequences from the Operation Mappings section of EXPRESSION into trees, by a simple use-def analysis.

The architecture-dependent optimizations make use of complex processor operations, such as MAC, as well as explore different implementations of certain operations (e.g., MULT x 2 y into SHL x y). They are implemented in manner similar to instruction selection. Below, we give the specification for some representative operation mappings.

```
(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))
	)
)
```

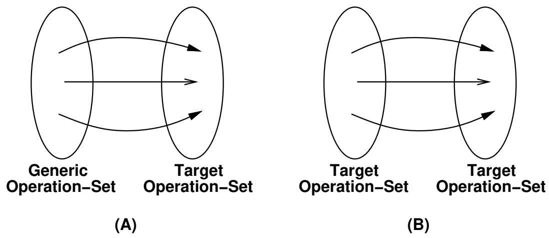

Figure 6: (A) Instruction selection and (B) Architecture specific optimizations

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. Each component also has a list of attributes (optional). The attributes can be any of the following:

- SUBCOMPONENTS: If the component is a compound component, specifies the list of subcomponents. (E.g. the *Execute* unit in the example below).
- LATCHES: If the component is a unit, specifies the list of latches that the unit is attached to.
- PORTS: The list of ports attached to this component.
- CONNECTIONS: The list of connections attached to this component.
- OPCODES: The list of opcode groups that this component accepts. (Note: If this attribute is NULL, then it means that the component does not make a distinction between opcodes.)
- TIMING: For multi-cycle or pipelined units, specifies the timing behavior. Timing can be specified on a per-opcode basis if necessary.
- CAPACITY: The number of operations that can be accepted by this component in a single cycle. The default is a single operation per cycle. (E.g. the *Decode* unit processes four operations per cycle in the example below).

Below, we give the specification for some representative components (selected from the architecture shown in Figure 4).

```
(SUBTYPE UNIT ExUnit CompExUnit AGUUnit FetchUnit DecodeUnit ArchUnit)
(SUBTYPE STORAGE Memory RegFile)
(SUBTYPE CONNECTION Bus CBox)
(SUBTYPE PORT ...)
(FetchUnit Fetch
    (LATCHES fetDecLatch)
    (PORTS fetProgMemPort)
    (CAPACITY 4)
)
(DecodeUnit Decode
    (LATCHES fetDecLatch decExLatch1 decExLatch2 decExLatch3 decExLatch4)
    (CAPACITY 4)
)
(ExUnit ALU()
    (LATCHES decExLatch1)
    (OPCODES alu_ops)
)
(ExUnit Mult()
    (LATCHES decExLatch2)
    (OPCODES mult_ops)
)
(AGUUnit AGU1()
    (LATCHES decExLatch3)
```

```
    (OPCODES pmove1_ops)
)
(AGUUnit AGU2()
    (LATCHES decExLatch4)
    (OPCODES pmove2_ops)
)
(CompExUnit Execute()
    (SUBCOMPONENTS ALU Mult AGU1 AGU2)
)
(ArchUnit Arch()
    (SUBCOMPONENTS Fetch Decode Execute)
)
(RegFile RFA()
    (CONNECTIONS C1 ALU_RFA)
)
(RegFile RFB()
    (CONNECTIONS C2 ALU_RFB)
)
(RegFile RFX()
    (CONNECTIONS C3 RFX_ALU1 RFX_ALU2 RFX_Mult1 RFX_Mult2)
)
(RegFile RFY()
    (CONNECTIONS C4 Mult_RFY)
)
(Storage X_mem()
    (CONNECTIONS C5)
)
(Storage Y_mem()
    (CONNECTIONS C6)
)
(Connection C1() C2() C3() C4() C5() C6() ALU_RFA() ALU_RFB() RFX_ALU1() RFX_ALU2() RFX_Mult1() RFX_Mult2() Mult_RFY())
(Bus X_bus()
    (CONNECTIONS C1 C3 C5 CrossBox)
)
(Bus Y_bus()
    (CONNECTIONS C2 C4 C6 CrossBox)
)
(CBox CrossBox())
```

5.2.5 Pipeline and Data-Transfer Paths Description

This subsection describes the net-list 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.

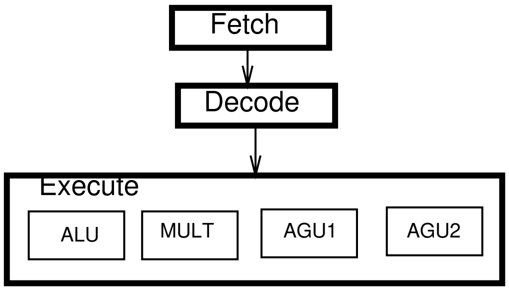

Figure 7: The pipeline for the example architecture in Figure 4

```
(PIPELINE FETCH DECODE EX) // 3 stage pipeline for architecture in Figure 4
(EX: 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)
    )
    (TYPE UNI
        (RFX ALU RFX_ALU1)
        (RFX ALU RFX_ALU2)
        (RFX Mult RFX_Mult1)
        (RFX Mult RFX_Mult2)
        (ALU RFA ALU_RFA)
        (ALU RFB ALU_RFB)
        (Mult RFY ALU_RFY)
    )
)
```

The pipeline paths and the data transfers represent the structural net-list information for the architecture. They can be generated automatically from a schematic capture tool.

When writing/generating this information, false paths may be present in the architecture. This problem is solved by the fact that besides the net-list, behavioral information is also present. False paths due to illegal operations will never be activated, since the illegal operations are omitted from the description. False paths due to illegal groups of operations (either in sequence or in parallel) can be tackled by either specifying the common resource used by the operations to raise a conflict in the reservation tables, or by explicitly specifying the group of operations as illegal.

5.2.6 Memory Subsystem

This subsection describes the properties of the components in the memory subsystem that are required by the memory-aware compiler optimizations.

In EXPRESSION, the net-list specification provides the connectivity between the various storage component and units. The attributes of each storage component (that are useful for the memory-aware compiler optimizations) are specified in this section. EXPRESSION can be used to describe diverse traditional and non-traditional memory systems. Non-traditional memory systems differ from the conventional, simple memory hierarchies in several ways. First, the complex organization of the components in these systems may result in a partitioned address space. The partitioned address space is captured using the *ADDRESS_RANGE* parameter associated with the relevant memory units. The *ACCESS_TIMES* parameter captures the latency information for individual memory components in the architecture. The latency associated with any level of the memory hierarchy can be easily computed using the hierarchy information (from the net-list) and the *ACCESS_TIMES* attribute of each component.

Second, they may contain novel components (e.g. SDRAM, Frame buffer, Stream buffer, etc). These components may allow for varying access times depending on the mode of access. The user can specify this feature as a list of access times in the *ACCESS_TIMES* attribute of the component.

The language provides certain pre–defined parameters (like *TYPE SIZE* etc). The *TYPE* parameter is used to identify each storage component. EXPRESSION contains certain predefined types like REGFILE, DRAM, CACHE, SRAM etc. New types can be easily added as a user-defined type. The user can also add new parameters in order to specify the features of novel components (for use by the compiler optimizations). Note that some parameters like the number and type of ports associated with each component are described in the Components Specification subsection.

In Figure 8, we consider an example architecture with one processor connected to off-chip memory. The processor contains four register files (RFA, RFB, RFX, RFY). The memory address space is divided between the SRAM (L1_sram) and DRAMs (X_mem, Y_mem). Further, the DRAMs are accessed through two set-associative caches (L1_X_cache, and L1_Y_cache).

The corresponding EXPRESSION specification is shown below.

From the specification it can be seen that the register files (RFA, RFB, RFX, RFY) contain two registers each, each register being 24 bits wide. The L1_sram is a fast memory of size 512 elements, each of size 12 bits. Assuming a word length of 24, the L1_sram can store 256 words. Hence the address range associated with it is 0–255. The access time for an access to the SRAM takes 2 cycles. The X_mem and Y_mem are DRAMs of size 512 words each, each word being 24 bits wide. The address range associated with X_mem is 256–767 and that with Y_mem is 768–1279. Both these memories have an access time of 10 cycles. Access to the X_mem is through L1_X_cache and that to the Y_mem is through the L1_Y_cache. Each of these caches have line width of 48 bits (i.e two 24 bit words), and size of 64 words. Hence each cache has 32 lines; each line containing 2 words. The L1_X_cache 2–way set associative while the L1_Y_cache is 4–way set associative. The address range of the L1_X_cache is the same as that of the X_mem (i.e 256–767) and the address range of the L1_Y_cache is the same as that of the Y_mem (768–1279). If the word requested by the processor is present in the one of the caches, it takes 1 cycle to retrieve that word. However, in case of a cache miss the word needs to be fetched from the corresponding DRAM, and the delay for this can be computed using the access times of the cache and the DRAM.

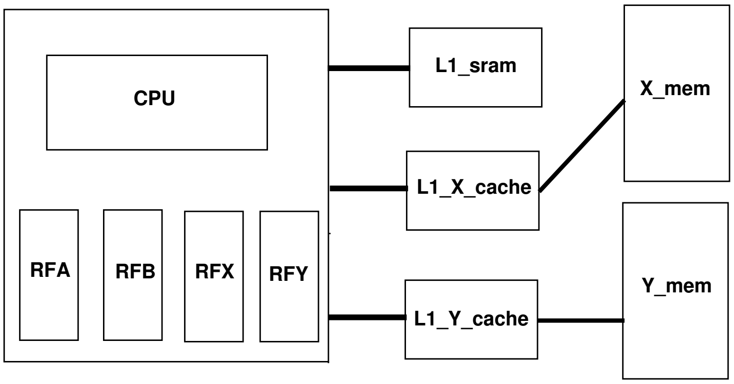

Figure 8: Memory hierarchy example

```
(STORAGE_PARAMETERS
    (RFA
        (TYPE REGFILE)
        (SIZE 2)
        (WIDTH 24)
    )
    (RFB
        (TYPE REGFILE)
        (SIZE 2)
        (WIDTH 24)
    )
    (RFX
        (TYPE REGFILE)
        (SIZE 2)
        (WIDTH 24)
    )
    (RFY
        (TYPE REGFILE)
        (SIZE 2)
        (WIDTH 24)
    )
    (L1_sram
        (TYPE SRAM)
        (SIZE 512)
        (WIDTH 12)
        (ADDRESS_RANGE 0 255)
        (ACCESS_TIMES 2)
    )
    (X_mem
```

```
        (TYPE DRAM)
        (SIZE 512)
        (WIDTH 24)
        (ADDRESS_RANGE 256 767)
        (ACCESS_TIMES 10)
    )
    (Y_mem
        (TYPE DRAM)
        (SIZE 512)
        (WIDTH 24)
        (ADDRESS_RANGE 768 1279)
        (ACCESS_TIMES 10)
    )
    (L1_X_cache
        (TYPE D_CACHE)
        (SIZE 64)
        (LINE 48)
        (ASSOCIATIVITY 2)
        (ADDRESS_RANGE 256 767)
        (ACCESS_TIMES 1)
    )
    (L1_Y_cache
        (TYPE D_CACHE)
        (SIZE 64)
        (LINE 48)
        (ASSOCIATIVITY 4)
        (ADDRESS_RANGE 768 1279)
        (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. Moreover, the user specifies the net-list information in order to generate a cycle-accurate structural simulator. This same information is also used to generate the reservation tables (without requiring the user to specify additional information explicitly for the purpose of generating reservation tables). Therefore, the reservation tables are generated at no extra cost for the user.

The input to the Reservation Tables generation algorithm is the EXPRESSION description containing the pipeline paths and the data transfers, as well as an operation. The output of the algorithm is the reservation table for that operation.

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 when total exploration time is critical (an exploration step usually generates one compiler, and runs it a few times to evaluate the current architecture), computing reservation tables on-the-fly is beneficial, since only few operations are referenced from any one application. Once the architecture has been fixed, and a quality tool-kit is needed (in the refinement phase, reservation tables are generated once, but many compilations are envisaged), the reservation tables are computed a priori to reduce the compilation time.

In the remainder of this section we use two examples to aid our explanation. First we present a sample architecture that illustrates the complexity involved in generating the reservation tables. We then use a simple architecture to illustrate the steps of the algorithm used to derive the reservation tables.

6.1 Sample Architecture Example

To intuitively explain the reservation tables generation algorithm and illustrate the complexity of the problem, we use the pipelined architecture in Figure 9, containing six functional units (ALU, SHIFT, MULT, ACC, LD/ST, BRANCH), two register files (RFA, RFB), and one memory (RAM). The bold lines represent pipeline paths, whereas the thin lines represent data transfer paths. The pipeline structure allows executing complex operations (e.g., MULT+ACC), as well as simple operations (e.g., MULT operation without ACC, by permitting bypassing of the ACC unit). The bypass paths (e.g., the bold arrow from MULT to the WR_BCK unit) allow bypassing a functional unit during the execution of an operation (e.g., executing only a multiply without accumulate, or executing an addition without shifting). The data transfer paths (the thin lines in Figure 9) allow the transfer of data between the register files, memory and the functional units.

In order to generate the reservation tables for a given operation, we determine for each stage of the pipeline which functional units, connections, busses and storage units are activated. We intuitively present the computation of reservation tables for the operation *MPYACC RFB0 RFA1 RFA2* $\rightarrow$ *RFA3*. Figure 10 shows the progression of this operation through the pipeline, and the resources used in each stage.

During execution, the operation progresses through the pipeline, and in each stage it may read some operands, compute some results on the functional units, and write back some operands. First we determine how the operation progresses through the pipeline. Based on the units used by the operation, after being fetched, it will go to one of the four decoders. We know that the operation uses the MULT and ACC units. Since the pipeline path containing MULT and ACC (i.e., FETCH $\rightarrow$ DEC2 $\rightarrow$ MULT $\rightarrow$ ACC $\rightarrow$ WR_BCK) uses the second decoder, after FETCH (Figure 10 a) the operation proceeds to DEC2 (Figure 10 b). Then it follows on to the MULT unit in stage 3, and reads the multiplication operands from RFA and RFB (Figure 10 c). In the fourth stage it continues with the ACC, and reads the operands from RFA (Figure 10 d). Finally, in the fifth stage it writes back the result.

Thus, by choosing a pipeline path, then traversing it and computing the data transfers needed at each stage, we determine the resource usage for the given operation, and construct the reservation table.

As shown in the example, the resource usage for an operation depends both on the operation's progression through the pipeline, and on the sources and destinations of the data transfers needed in each pipeline stage. Thus, deriving detailed reservation tables requires traversing the pipeline and finding the units involved in the computation, determining the data transfers needed, their initiation time and duration, as well as the route taken by each data transfer.

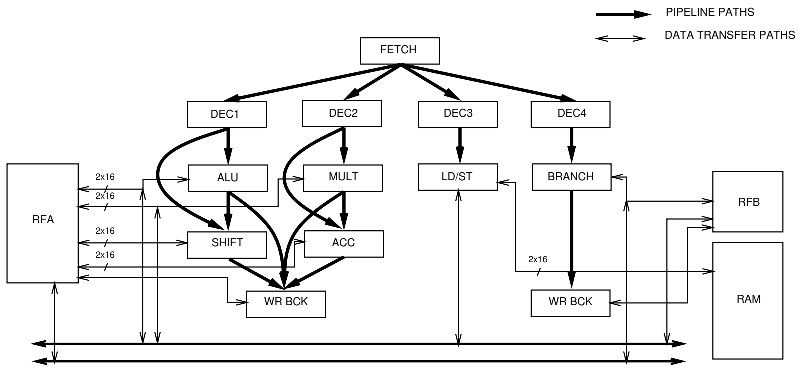

Figure 9: Pipeline and data transfers structure for a complex architecture

In Figure 11 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').

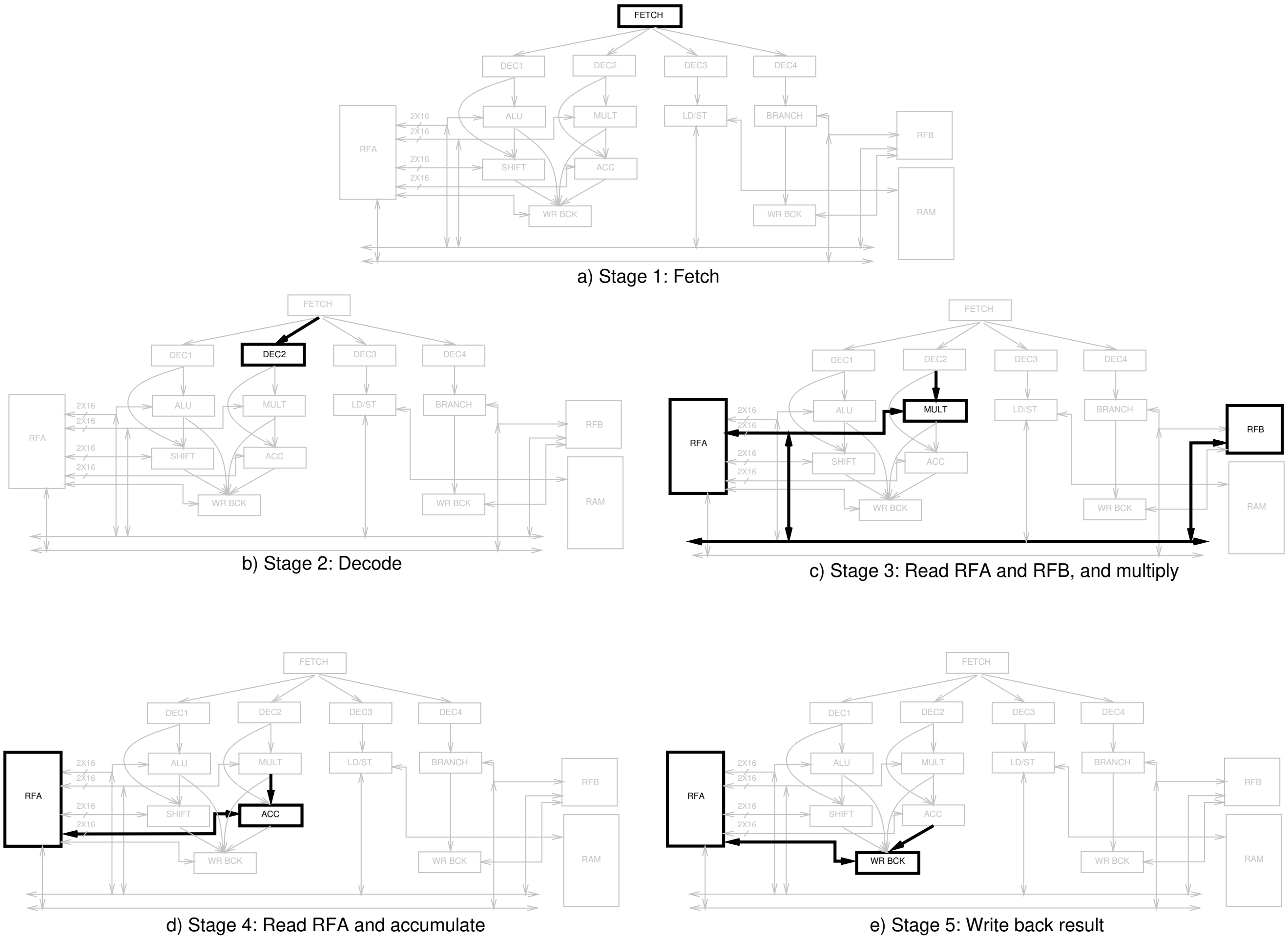

Figure 10: Pipeline and data transfer paths selected in consecutive pipeline stages for *MPYACC RFB0 RFA1 RFA2* $\rightarrow$ *RFA3*

Steps needed for Reservation Table Generation.	Example to illustrate the Reservation Table Generation.
Input: Operation Op	Input: *pmove1 (AGU1) X_mem[100] RFB*
Output: Reservation Table (R.T.) for the operation.	Output: /* Reservation Table for the operation */ /* Cyc stands for Cycle */
A. Find pipeline path which contains Op's F.U.	Pipeline path - Fetch; Decode; AGU1;
B. Generate partial R.T. from the pipeline path.	Partial R.T. - Cyc1: Fetch; Cyc2: Decode; Cyc3: AGU1;
C. **For** each data-transfer initiatedby Op	
Determine the source and sinkcomponents.	Source: X_mem, Sink: RFB
Find transfer segments which implement. the data-transfer.	(C5 Xbus) (Cross box) (Ybus C2)
Add objects in the segment to R.T.	R.T. - Cyc1 ... Cyc3: AGU1, X_mem, C5, Xbus, Cross Box, Ybus C2 RFB
/* Output the completed reservation table */	

Figure 11: The steps in Reservation Table generation illustrated with an example (also refer Figure 4)

7 Support for Design Space Exploration (DSE)

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 case (e.g. for novel memory organization and hierarchies) they are not feasible.

In order to give a flavor of the ease of specification and modification using EXPRESSION, we explore how easy it is to make local changes to the architecture and to add/delete operations and compare this with equivalent changes in MDes. The reason in choosing MDes for the comparison, is that it is closest to our approach, and seems to provide the best support for design space exploration, by capturing both behavioral and structural information.

We illustrate the differences 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. The complete specifications for MDes and EXPRESSION can be viewed at:
http://www.ics.uci.edu/~ahalambi/EXPRESS/specification.html.

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.

7.1 Support for verification of the EXPRESSION specification

As mentioned before, EXPRESSION is a mixed-level language wherein the user specifies both the structure and the instruction set of the system. Furthermore, with the complex systems of today, the user is required to specify hundreds of instructions and architectural components. Thus, the ADLs designed to specify these systems, to be useful, must allow for conciseness in specification and, more importantly, should contain mechanisms that check for correctness and consistency in specification. Often times, there is even a trade-off between conciseness of specification, and language mechanisms that guide correctness checking.

We believe that the problems posed in the preceding paragraph will gain even further importance in the future as systems grow in complexity and time-to-market shortens. Further, in order to perform rapid DSE, the designer needs tools that automatically perform (at least a limited form of) formal verification of the specification. This problem is a very hard problem and is a hot-bed of research activity in the ADL community. In designing EXPRESSION, we considered some limited forms of formal verification that show promise in catching errors that may creep up in the specification. They are:

Property checking: Verify if certain user-defined properties are upheld. (e.g. the number of ports in each register file is less than a certain number).

Connectivity checking: Verify if the net-list specification (i.e. both data-path and pipeline) is connected.

Consistency checking: Verify for consistency between the structure and the instruction-set specification. (e.g. each opcode group described in the instruction-set should be associated with at least one unit).

Completeness checking: Verify if the specification contains enough information in order to perform compilation and simulation. (e.g. The Operation Mappings section should contain at least one entry for each generic opcode.)

As briefly mentioned above, EXPRESSION possesses some features that make verification feasible. This capacity, coupled with its flexibility in terms of making changes to the specification, makes it a very attractive language for DSE. In the succeeding paragraphs we elaborate on the modifiability feature of EXPRESSION.

7.2 Support for modification of the EXPRESSION specification

During DSE, the system designer would like to make structural modifications (e.g., adding/deleting components, connections), as well as behavioral modifications (e.g., changing the instruction set). In the following we present four modifications to the architecture, and for each show the changes needed to adapt the EXPRESSION and MDes descriptions. The first three modifications alter the structure of the example architecture, by splitting a register file, adding connections and busses. The fourth modification alters the instruction set, by adding a new operation.

7.2.1 Splitting a register file

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 12.

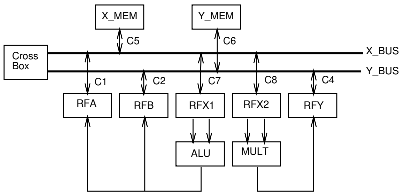

Figure 12: The modified example with RFX split into RFX1 and RFX2, connection C3 deleted and connections C7 and C8 added

This change in the architecture impacts the resource constraints specification of both MDes and EXPRESSION. Shown below are the additions made to the specifications of both MDes and EXPRESSION in order to incorporate the changes to the example architecture (Figure 12).

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

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 (9 lines added), whereas in EXPRESSION the changes were more localized, requiring modifications to 2 sections (2 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.

7.2.2 Adding a connection

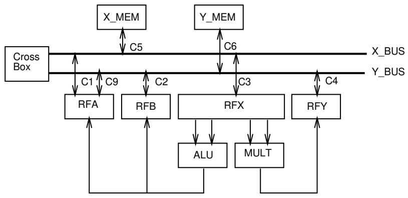

Figure 13: The modified example with connection C9 added between RFA and Y_bus

In this exploration example, in order to make transfers between RFA and RFB easier (assuming there were many conflicts on those transfers) we add a new connection c9 between RFA and the Y_bus (Figure 13). Again, in MDes we need to make changes in many levels of the hierarchy for the reservation tables. We needed to reanalyze many parts of the MDes description to determine if they were affected by the change. In EXPRESSION, due to the architecture-centered representation, the information related to connections is more local, and furthermore requires changes to only 2 lines of the description.

Changes to MDes:

```
SECTION Resource{
...
C9();
}
SECTION Resource_Usage{
// All resources are used in only one cycle.
$for (... C9){
RU_${CASE} (use(${CASE}) time(0));
}
}
SECTION Resource_Units{
...
RFA_unit2 (use(RFA C9 Y_bus));
}
SECTION Table_Option{
// Specifies options for the reservation tables.
...
any_Y_reg_transfer (one_of( RFA_unit2));
}
```

Changes to EXPRESSION:

```
(Connection ... C9())
(DTPATHS
(TYPE BI
...
(RFA Y_bus C9)
)
)
```

7.2.3 Adding a bus and 2 connections

During exploration, in order to allow transfers between RFA and RFB in parallel with transfers between the X_bus and Y_bus, we add the bus G_bus connected to the Cross-box, and communicating with either X_bus or Y_bus, and the connections c10 and c11 between RFA, RFB and G_bus respectively (Figure 14). To reflect this change in the descriptions, changes to MDes are spread out in the description at different levels in the hierarchy

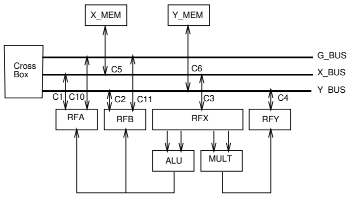

Figure 14: The modified example with a bus and 2 connections added

(due to the instruction-centered view, as opposed to architecture-centered view). Changes to EXPRESSION are local to the net-list, requiring the modification of only 6 lines of code, as opposed to 16 in MDes.

Changes to MDes:

```
SECTION Resource{
...
C10(); C11();
G_bus();
}
SECTION Resource_Usage {
// All resources are used in only one cycle.
$for (... C10 C11 G_bus)
{
RU_${CASE} (use(${CASE}) time(0));
}
}
SECTION Resource_Units{
// Combine the connections and storages where the usage of one
// implies the usage of the other.
...
RFA_unit2 (use(RFA C10 G_bus));
RFB_unit2 (use(RFA C11 G_bus));
}
SECTION Table_Option{
// Specifies options for the reservation tables.
any_X_reg_transfer (one_of(RFA_unit1 ));
any_Y_reg_transfer (one_of(RFB_unit2 ));
}
SECTION Reservation_Table{
// Show how resources are used as operations execute.
...
RT_PMOVE1_XtoRFB (use (RU_AGU1 X_mem_unit RU_G_bus RFB_unit2));
RT_PMOVE1_YtoRFA (use (RU_AGU1 Y_mem_unit RU_G_bus RFA_unit2));
RT_PMOVE1_RFA_Yreg (use(RFA_unit2 RU_G_bus any_Y_reg_transfer));
RT_PMOVE1_RFB_Xreg (use(RFB_unit2 RU_G_bus any_X_reg_transfer));
RT_PMOVE1_RFA_RFB (use(RFA_unit2 RU_G_bus RFB_unit2));
}
SECTION Scheduling_Alternative{
// Each scheduling alternative is linked to one reservation table.
$for (CASE in ... PMOVE1_XtoRFB PMOVE1_YtoRFA PMOVE1_RFA_Yreg
PMOVE1_RFB_Xreg PMOVE1_RFA_RFB){
ALT_${CASE} (resv(RT_${CASE}));
}
}
SECTION Operation{
$for (CASE in ... PMOVE1_XtoRFB PMOVE1_YtoRFA
PMOVE1_RFA_Yreg PMOVE1_RFB_Xreg
PMOVE1_RFA_RFB){
OP_${CASE} (alt(ALT_${CASE}));
}
}
```

Changes to EXPRESSION:

```
(Connection ... C10() C11())
```

```
(Bus ... G_bus())
(DTPATHS
(TYPE BI
...
(RFA G_bus C10)
(RFB G_bus C11)
(X_bus G_bus Cbox)
(Y_bus G_bus Cbox)
)
)
```

7.2.4 Adding an operation

In the following exploration step we add the SHL opcode to the ALU functional unit. We assume that SHL is also a 1 cycle operation (just like ADD, SUB), which is similar to the Motorola 56000 processor. Even though EXPRESSION is more structure-centered, to the instruction set were not much more cumbersome than in MDes. In fact they were fairly local for both descriptions.

The information mentioned for both the MDes and EXPRESSION examples is limited to the reservation tables only. Information for instruction selection and register allocation is omitted from both descriptions (e.g., for instruction selection, the information regarding the mapping of generic compiler operations to SHL has to be captured, while for register allocation the possible source/destinations of the operands are needed). The information for instruction selection and register allocation is captured in a similar manner in EXPRESSION and MDes.

Changes to MDes:

Due to the fact that SHL is a 1 cycle operation just like ADD and SUB, no extra reservation table options are needed to be added in MDes. The same one used for ADD and SUB can be used also for SHL.

```
SECTION Operation
{
$for (CASE in ... SHL)
{
OP_${CASE} (alt(ALT_ALUOP));
}
}
```

Changes to EXPRESSION:

In EXPRESSION the only addition for reservation tables is including the SHL operation in the opcodes supported by the ALU functional unit.

```
(OP_GROUP alu_ops
(OPCODE ... SHL)
)
```

Table 1 shows the number of lines of code and the number of sections requiring modifications in the MDes and EXPRESSION descriptions to account for the architectural and behavioral changes presented above. Columns 2 and 3 show the number of lines of code needing modification, while columns 4 and 5 show the number of different sections where modifications are required. For most of the cases, MDes requires two- to four-fold more lines changed (e.g., Register File Split). In the last exploration step adding an operation, the number of lines and sections requiring modification is very low for both approaches. As can be seen from the above examples, structural modifications are better accommodated by EXPRESSION, while behavioral changes present no significant difference. Thus, EXPRESSION seems to be better suited as a language for rapid DSE.

8 Summary

In this report we presented 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,

Modification	Lines of Code		Number of Sections	
	MDes	EXPRESSION	MDes	EXPRESSION
Split RF	9	2	7	2
Add Connect.	4	2	4	2
Add Bus and Conn's	16	6	7	3
Add Operation	1	2	1	2
Total	30	12	19	9

Table 1: Number of lines and sections requiring modification in MDes and EXPRESSION

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.

Due to the way EXPRESSION captures both the structural and behavioral information of the architecture, making structural modifications to the design during exploration requires only local changes to the description. Behavioral changes are also easily supported. Features such as reservation tables generation, non-traditional memory subsystem specification, and combining of the simulator and compiler information into one description are not present in any other ADL, but they are vital to support fast Design Space Exploration. We presented a detailed experiment comparing a range of exploration steps performed for an EXPRESSION and an MDes description for an example architecture.

We also proposed 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. We present in detail the algorithm which generates the Reservation Tables from the EXPRESSION specification.

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 the memory-aware compiler optimizations.

Table 2 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 net-list. 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 [20]), DSP (Motorola 56K [15]) and Superscalar (MIPS R10000) have been implemented. EXPRESSION is currently used to drive several projects at U.C. Irvine, including MEMOREX (a memory estimation and exploration environment) [5], [16] 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.

9 Acknowledgments

This research was supported in part through grants from NSF (MIP-9708067) and ONR (N00014-93-1-1348). We would like to gratefully acknowledge Nick Savoiu who contributed to the EXPRESS/EXPRESSION project through his ideas and dedicated work.

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

Table 2: Comparison between different ADLs

10 References

[1] J. C. Gyllenhaal et al. The MDES user manual. Technical report, Trimaran Release: http://www.trimaran.org, 1998.

[2] Leupers et al. Retargetable generation of code selectors from HDL processor models. In *Proceedings of European Design & Test Conference*, Paris, France, March 1997.

[3] C. Fraser, D. Hanson, and R. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, 1992.

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

[5] P. Grun, F. Balasa, and N. Dutt. Memory size estimation for multimedia applications. In *Proc. CODES/CACHE '98*.

[6] J. C. Gyllenhaal. A machine description language for compilation. Master's thesis, Department of Electrical Engineering, University of Illinois at Urbana-Champaign, 1994.

[7] J. C. Gyllenhaal, W. W. Hwu, and B. R. Rau. Optimization of machine descriptions for efficient use. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, December 1996.

[8] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *Proceedings of Design Automation Conference*, Anaheim, CA, May 1997.

[9] S. Hanono and S. Devadas. Instruction selection,resource allocation,and scheduling in the AVIV retargetable code generator. In *Proceedings of 35th Design Automation Conference*, San Fransisco, CA, June 1998.

[10] A. Inoue, H. Tomiyama, E. Nurprasetyo, and H. Yasuura. A programming language for processor based embedded systems. In *Proceedings of Asia Pacific Conference on Hardware Description Languages (APCHDL'98)*, pages 89–94, July 1998.

[11] L. Kohn and G. Efland. Making personal video a reality. *Byte*, pages 59–60, December 1997.

[12] D. Lanneer, J. Van Praet, A. Kifli, K.Schoofs, W.Geurts, F. Thoen, and G. Goosens. *CHESS: Retargetable Code Generation for Embedded DSP Processors. In P. Marwedel and G. Goosens, editors, Code Generation for Embedded Processors, chapter 5, pages 85-102.* Kluwer Academic Publishers, Boston, Massachusetts, 1997.

[13] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1):1–36, January 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 '97*.

[16] P.R. Panda, N.D. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, Boston, Massachusetts, 1998.

[17] P. Paulin, C. Liem, T. May, and S. Sutarwala. Flexware: A flexible firmware development environment for embedded systems. In *Proceedings of Dagstuhl Code Generation Workshop*, August 1994.

[18] C. Siska. A processor description language supporting retargetable multi-pipeline dsp program development tools. In *Proceedings of the 11th Interational Symposium on System Synthesis (ISSS-98)*, December 1998.

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

[20] V. Zivojnovic, S. Pees, and H. Meyr. LISA - machine description language and generic machine model for HW/SW co-design. In *IEEE*, December 1996.

A BNF Description of EXPRESSION

A.1 Operations Specification

```
<operation_info> : <op_group_list> [<var_groups>] [<operand_semantics_list>]
<op_group_list> : <op_group_list> <op_group> | <op_group>
<op_group> : '(' OP_GROUP <op_group_NAME> <op_list> ')'
<op_list> : <op_list> <op> | <op>
<op> : '(' OPCODE <opcode_NAME>
  '(' OP_TYPE <op_type> ')'
  '(' OPERANDS <operand_list> ')'
  '(' BEHAVIOR <op_semantics> ')'
        ')'
<op_type> : DATA_OP || CONTROL_OP || FLOW_OP
<operand_list> : <operand_list> <one_operand> | <one_operand>
<one_operand> : '(' <operand_type> <var> ')'
<operand_type> : SRC<num> || DST<num>
<var> : <var_group_NAME> <operand_semantics>
<operand_semantics> : <operand_semantics_group_NAME> || <operand_semantics_operator>
<op_semantics> : % the semantics of the operation.
<var_groups> : '(' VAR_GROUPS <var_group_list> ')'
<var_group_list> : <var_group_list> <var_group> || <var_group>
<var_group> : '(' <var_group_NAME> <var_list> ')'
<var_list> : <var_list> '|' <component_NAME> || <component_NAME>
<operand_semantics_list> : '(' OPERAND_SEMANTICS <operand_sem_groups> ')'
<operand_sem_groups> : <operand_sem_groups> <operand_sem_group> || <operand_sem_group>
<operand_sem_group> : '(' <operand_semantics_group_NAME> <operand_sem_list> ')'
<operand_sem_list> : <operand_sem_list> '|' <operand_sem_operator> || <operand_sem_operator>
<op_group_NAME>, <opcode_NAME> : identifier
<var_group_NAME>, <operand_semantics_group_NAME> : identifier
```

A.2 Instruction Description

```
<instruction_info> : '(' INSTR <word_length> <slot_list> ')'
<word_length> : '(' WORDLEN <num> ')'
<slot_list> : '(' SLOTS <s_list> ')'
<s_list> : <s_list> <slot> | <slot>
<slot> : '(' <s_type> <bit_width> <s_unit> ')'
<s_type> : '(' TYPE <type_name> ')'
<type_name> : DATA || CONTROL || FLOW
<bit_width> : '(' BITWIDTH <num> ')'
<unit> : '(' UNIT <component_NAME> ')'
```

A.3 Operation Mappings

```
<operation_mapping_info> : '( OP_MAPPING <op_mapping_list> ')'
<op_mapping_list> : <op_mapping_list> <op_mapping> || <op_mapping>
<op_mapping> : '(' <domain_ops> <range_ops> ')'
<domain_ops> : '(' <domain_name> <oper_list> ')'
<range_ops> : '(' <range_name> <oper_list> ')'
<oper_list> : <oper_list> <oper> || <oper>
<oper> : '(' <opcode_NAME> <map_var_list> ')'
<map_var_list> : <map_var_list> <map_var> | <map_var>
<map_var> : <map_var_id> [<operand_semantics>]
```

```
<map_var_id> : <map_var_NAME> || <immediate_num>
% <operand_semantics> : see operation information section.
<domain_name> : GENERIC || TARGET
<range_name> : TARGET
<map_var_NAME> : identifier
<immediate_num> : <num>
```

A.4 Components Specification

```
<arch_components_info> : <sub_type_decls> <comp_instantiation>
<sub_type_decls> : <sub_type_decls> <sub_type_decl> || <sub_type_decl>
<sub_type_decl> : '(' SUBTYPE <base_type> <subtype_list> ')'
<base_type> : UNIT || STORAGE || CONNECTION || PORT
<subtype_list> : <subtype_list> <subtype> || <subtype>
<subtype> : <comp_type_NAME>
<comp_instantiation> : '(' <comp_type_NAME> <comp_list> ')'
<comp_list> : <comp_list> <one_comp> || <one_comp>
<one_comp> : <component_NAME> '(' [<initialization>] ')' [<comp_attributes>]
<comp_atrributes> : [<sub_comps>] [<latches>] [<fan_in_out>] [<opcodes>] [<timing>] [<capacity>]
<sub_comps> : '(' SUBCOMPONENTS <sub_comp_list> ')'
<sub_comp_list> : <sub_comp_list> <sub_comp> || <sub_comp>
<sub_comp> : <component_NAME>
<latches> : '(' LATCHES <latch_list> ')'
<latch_list> : <latch_list> <latch> || <latch>
<latch> : <component_NAME>
<fan_in_out> : <ports> || <connections>
<ports> : '(' PORTS <port_list> ')'
<port_list> : <port_list> <port> || <port>
<port> : <component_NAME>
<connections> : '(' CONNECTIONS <connection_list> ')'
<connection_list> : <connection_list> <connection> || <connection>
<connection> : <component_NAME>
<opcodes> : '(' OPCODES <opcode_group_list> ')'
<opcode_group_list> : <opcode_group_list> <one_opcode_group> || <one_opcode_group>
<one_opcode_group> : <op_group_NAME> || <opcode_NAME>
<timing> : '(' TIMING <timing_info> ')'
<timing_info> : '(' <opcode_group_list> <cycle_num_list> ')'
<cycle_num_list> : '(' <cycle_nums> ')'
<cycle_nums> : <cycle_nums> <cycle_num> || <cycle_num>
<cycle_num> : C<num>
<capacity> : '(' CAPACITY <num> ')'
<comp_type_NAME>, <component_NAME> : identifier
% <op_group_NAME>, <opcode_NAME> : // see operations info section
```

A.5 Pipeline and Data Transfer Paths

```
<pipe_dt_paths_info> : <pipeline_info> <dt_paths_info>
<pipeline_info> : <pipeline_list> [<pipe_comp_unit_list>]
<pipeline_list> : <pipeline_list> <pipeline> || <pipeline>
<pipeline> : '(' PIPELINE <pipe_unit_name_list> ')'
<pipe_unit_name_list> : <pipe_unit_name_list> ';' <pipe_stage> || <pipe_stage>
<pipe_stage> : <pipe_stage_unit_list>
<pipe_stage_unit_list> : <pipe_stage_unit_list> ',' <pipe_unit> || <pipe_unit>
<pipe_unit> : <pipe_comp_unit_NAME> || <pipe_name>
```

```
<pipe_comp_unit_list> : <pipe_comp_unit_list> <pipe_comp_unit> || <pipe_comp_unit>
<pipe_comp_unit> : '(' <pipe_comp_unit_NAME> ':' <pipe_units> ')'
<pipe_units> : <pipe_units> '|' <pipe_name> || <pipe_name>
<dt_paths_info> : '(' DTPATHS <dt_paths_list> ')'
<dt_paths_list> : <dt_paths_list> <dt_paths> || <dt_paths>
<dt_paths> : '(' TYPE <t_type> <dt_list> ')'
<t_type> : UNI || BI
<dt_list> : <dt_list> <dt> || <dt>
<dt> : '(' <dt_start> <dt_end> <dt_units_list> ')'
<dt_units_list> : <dt_units_list> ';' <dt_units> || <dt_units>
<dt_units> : <dt_units> ',' <dt_unit> || <dt_unit>
<pipe_comp_unit_NAME> : <component_NAME> % // see architecture components section.
<dt_unit>, <dt_start>, <dt_end> : <component_NAME>
```

A.6 Memory Subsystem

```
<storage_info> : '(' STORAGE_PARAMETERS <storage_list> ')'
<storage_list> : <storage_list> <storage_unit> || <storage_unit>
<storage_unit> : '(' <storage_name> <storage_type>
[<st_size>]
[<st_width>]
[<st_line>]
[<st_associativity>]
[<st_addressrange>]
[<st_accesstimes>]
 ')'
<storage_type> : '(' TYPE <st_type> ')'
<st_type> : I_CACHE || D_CACHE || REGFILE || MEMORY || ...
<st_size> : '(' SIZE <num> ')'
<st_width> : '(' WIDTH <num> ')'
<st_line> : '(' LINE <num> ')'
<st_associativity> : '(' ASSOCIATIVITY <num> ')'
<st_addressrange> : '(' ADDRESSRANGE <num> ')'
<st_accesstimes> : '(' ACCESSTIMES <access_time_list> ')'
<access_time_list> : <access_time_list> <access_time> || <access_time>
<access_time> : <num>
<storage_name> : <component_NAME> % // See architecture components section
```