Processor Modeling and Design Tools

Prabhat Mishra
Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611, USA
prabhat@cise.ufl.edu

Nikil Dutt
Center for Embedded Computer Systems
Donald Bren School of Information and Computer Sciences
University of California, Irvine, CA 92697, USA
dutt@uci.edu

Abstract

This chapter covers state-of-the art specification languages, tools, and methodologies for processor development in academia as well as industry. Time-to-market pressure coupled with short-product lifetimes create a critical need for design automation in processor development. The processor is modeled using a specification language such as Architecture Description Language (ADL). The ADL specification is used to generate various tools (e.g., simulators, compilers and debuggers) to enable exploration and validation of candidate architectures. The goal is to find the best possible processor architecture for the given set of application programs under various design constraints such as cost, area, power and performance. The ADL specification is also used to perform various design automation tasks including hardware generation and functional verification of the processor.

1 Introduction

Computing is an integral part of daily life. We encounter two types of computing devices everyday: desktop based computing devices and embedded computer systems. Desktop based computing systems encompass traditional "computers", including personal computers, notebook computers, workstations, and servers. Embedded computer systems are ubiquitous - they run the computing devices hidden inside a vast array of everyday products and appliances such as cell phones, toys, handheld PDAs, cameras, and microwave ovens. Both types of computing devices use programmable components such as processors, coprocessors and memories to execute the application programs. These programmable components are also referred as *programmable architectures*. Figure 1 shows an example embedded system with programmable architectures. Depending on the application domain, the embedded system can have application specific hardwares, interfaces, controllers, and peripherals. The complexity of the programmable architectures is increasing at an exponential rate due to technological advances as well as demand for realization of ever

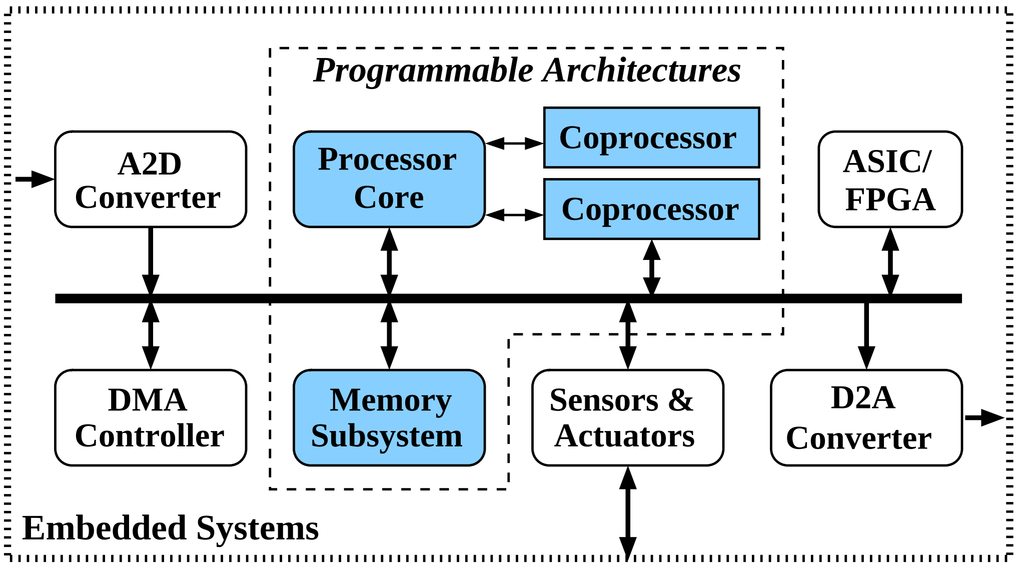

Figure 1. An example embedded system

more complex applications in communication, multimedia, networking, and entertainment. Shrinking time-to-market coupled with short product lifetimes create a critical need for design automation of increasingly sophisticated and complex programmable architectures.

Modeling plays a central role in design automation of processors. It is necessary to develop a specification language that can model complex processors at a higher level of abstraction and also enable automatic analysis and generation of efficient prototypes. The language should be powerful enough to capture high-level description of the programmable architectures. On the other hand, the language should be simple enough to allow correlation of the information between the specification and the architecture manual.

Specifications widely in use today are still written informally in natural language like English. Since natural language specifications are not amenable to automated analysis, there are possibilities of ambiguity, incompleteness, and contradiction: all problems that can lead to different interpretations of the specification. Clearly, formal specification languages are suitable for analysis and verification. Some have become popular because they are input languages for powerful verification tools such as a model checker. Such specifications are popular among verification engineers with expertise in formal languages. However, these specifications are not acceptable by designers and other tool developers. Therefore, the ideal specification language should have formal (unambiguous) semantics as well as easy correlation with the architecture manual.

Architecture Description Languages (ADL) have been successfully used as a specification language for processor development. The ADL specification is used to perform early exploration, synthesis, test generation, and validation of processor-based designs as shown in Figure 2. The ADL specification can also be used for generating hardware prototypes [42, 35]. Several researches have shown the usefulness of ADL-driven generation of functional test programs [47] and test interfaces [36]. The specification can also be used to generate device drivers for real-time operating systems [56].

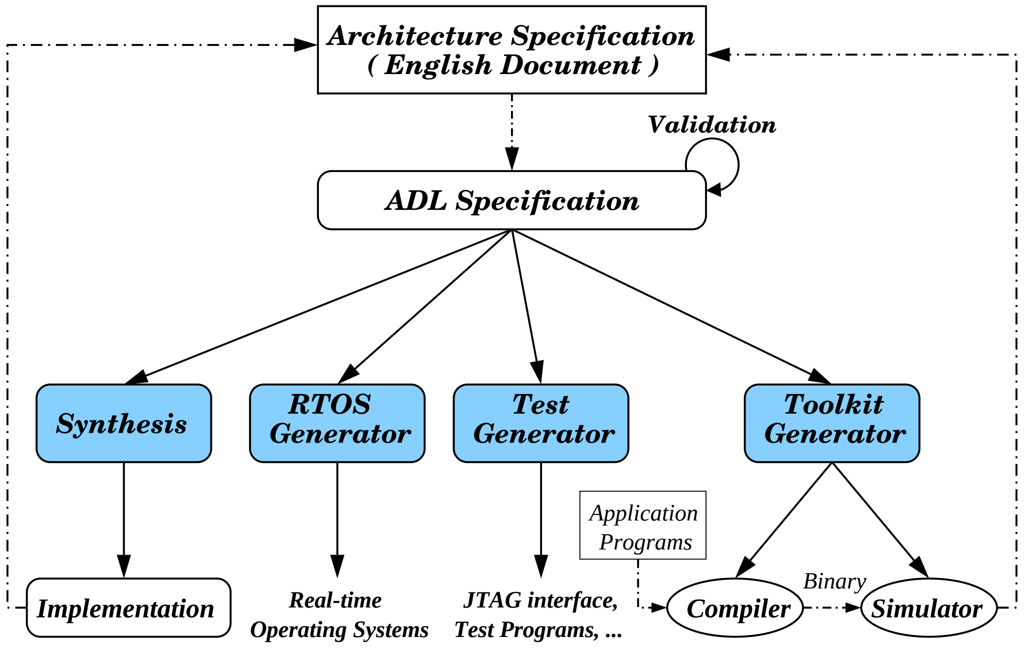

Figure 2. ADL-driven exploration, synthesis, and validation of programmable architectures

The rest of the chapter is organized as follows. Section 2 describes processor modeling using ADLs. Section 3 presents ADL-driven methodologies for software toolkit generation, hardware synthesis, exploration, and validation of programmable architectures. Finally, Section 4 concludes the chapter.

2 Processor Modeling using ADLs

The phrase "Architecture Description Language" (ADL) has been used in context of designing both software and hardware architectures. Software ADLs are used for representing and analyzing software architectures ([34], [50]). They capture the behavioral specifications of the components and their interactions that comprises the software architecture. However, hardware ADLs capture the structure (hardware components and their connectivity), and the behavior (instruction-set) of processor architectures. The concept of using machine description languages for specification of architectures has been around for a long time. Early ADLs such as ISPS [30] were used for simulation, evaluation, and synthesis of computers and other digital systems. This section describes contemporary hardware ADLs.

This section is organized as follows. First, it tries to answer why ADLs (not other languages) for modeling and specification. Next, it surveys contemporary ADLs to compare their relative strengths and weaknesses in the context of processor modeling and ADL-driven design automation.

2.1 ADLs and Other Languages

How do ADLs differ from programming languages, hardware description languages, modeling languages, and the like? This section attempts to answer this question. However, it is not always possible to answer the following question: Given a language for describing an architecture, what are the criteria for deciding whether it is an ADL or not?

In principle, ADLs differ from programming languages because the latter bind all architectural abstractions to specific point solutions whereas ADLs intentionally suppress or vary such binding. In practice, architecture is embodied and recoverable from code by reverse engineering methods. For example, it might be possible to analyze a piece of code written in **C** and figure out whether it corresponds to *Fetch* unit or not. Many languages provide architecture level views of the system. For example, C++ offers the ability to describe the structure of a processor by instantiating objects for the components of the architecture. However, C++ offers little or no architecture-level analytical capabilities. Therefore, it is difficult to describe architecture at a level of abstraction suitable for early analysis and exploration. More importantly, traditional programming languages are not natural choice for describing architectures due to their inability for capturing hardware features such as parallelism and synchronization.

ADLs differ from modeling languages (such as UML) because the later are more concerned with the behaviors of the whole rather than the parts, whereas ADLs concentrate on representation of components. In practice, many modeling languages allow the representation of cooperating components and can represent architectures reasonably well. However, the lack of an abstraction would make it harder to describe the instruction-set of the architecture. Traditional Hardware Description Languages (HDL), such as VHDL and Verilog, do not have sufficient abstraction to describe architectures and explore them at the system level. It is possible to perform reverse-engineering to extract the structure of the architecture from the HDL description. However, it is hard to extract the instruction-set behavior of the architecture. In practice, some variants of HDLs work reasonably well as ADLs for specific classes of programmable architectures.

There is no clear line between ADLs and non-ADLs. In principle, programming languages, modeling languages, and hardware description languages have aspects in common with ADLs, as shown in Figure 3. Languages can, however, be discriminated from one another according to how much architectural information they can capture and analyze. Languages that were born as ADLs show a clear advantage in this area over languages built for some other purpose and later co-opted to represent architectures.

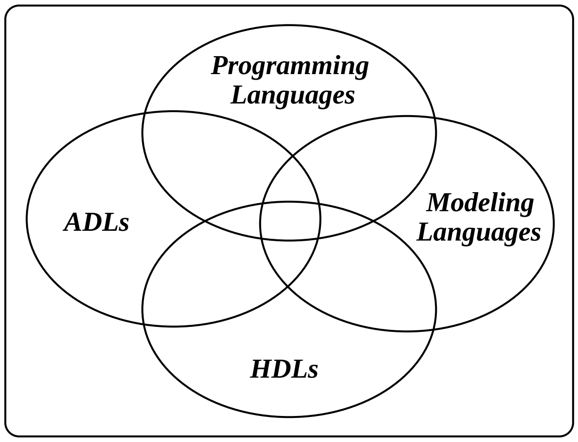

Figure 3. ADLs versus non-ADLs

2.2 Contemporary ADLs

This section briefly surveys some of the contemporary ADLs in the context of processor modeling and design automation. There are many comprehensive ADL surveys available in the literature including ADLs for retargetable compilation [61], programmable embedded systems [46], and SOC design [16].

Figure 4 shows the classification of architecture description languages (ADLs) based on two aspects: *content* and *objective*. The content-oriented classification is based on the nature of the information an ADL can capture, whereas the objective-oriented classification is based on the purpose of an ADL. Contemporary ADLs can be classified into six categories based on the objective: simulation-oriented, synthesis-oriented, test-oriented, compilation-oriented, validation-oriented, and operating system (OS) oriented.

ADLs can be classified into four categories based on the nature of the information: structural, behavioral, mixed, and partial. The structural ADLs capture the structure in terms of architectural components and their connectivity. The behavioral ADLs capture the instruction-set behavior of the processor architecture. The mixed ADLs capture both structure and behavior of the architecture. These ADLs capture complete description of the structure or behavior or both. However, the partial ADLs capture specific information about the architecture for the intended task. For example, an ADL intended for interface synthesis does not require internal structure or behavior of the processor.

Traditionally, structural ADLs are suitable for synthesis and test-generation. Similarly, behavioral ADLs are suitable for simulation and compilation. It is not always possible to establish a one-to-one correspondence between content-based and objective-based classification. For example, depending on the nature and amount of information captured, partial ADLs can represent any one or more classes of the objective-based ADLs. This section presents the survey using content-based classification of ADLs.

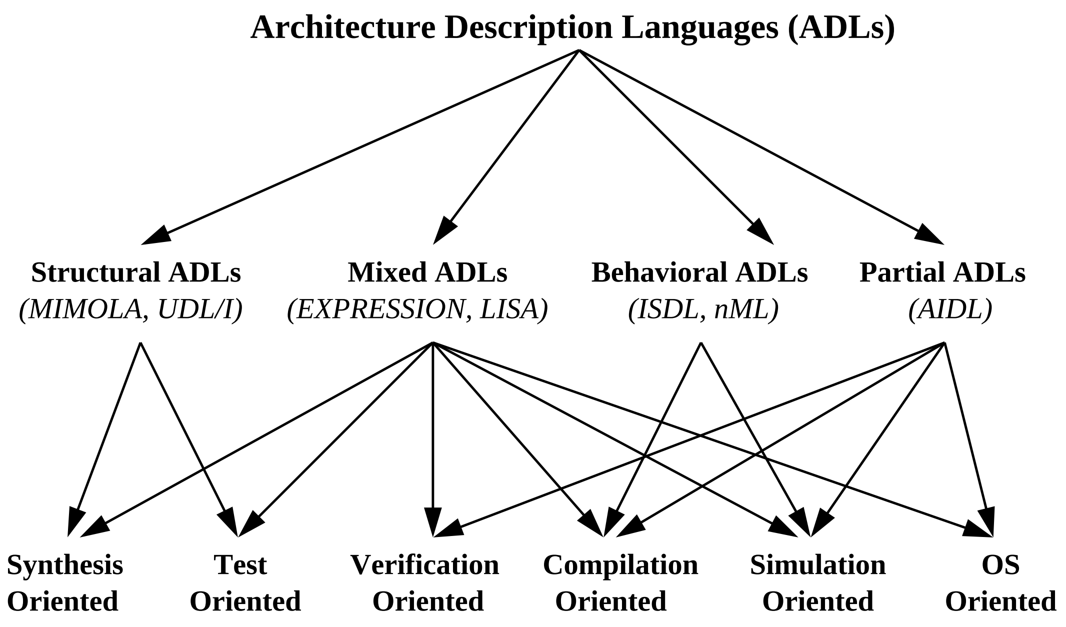

Figure 4. Taxonomy of ADLs

2.2.1 Structural ADLs

ADL designers consider two important aspects: level of abstraction versus generality. It is very difficult to find an abstraction to capture the features of different types of processors. A common way to obtain generality is to lower the abstraction level. Register transfer level (RT-level) is a popular abstraction level - low enough for detailed behavior modeling of digital systems, and high enough to hide gate-level implementation details. Early ADLs are based on RT-level descriptions. This section briefly describes a structural ADL: MIMOLA [53].

MIMOLA

MIMOLA [53] is a structure-centric ADL developed at the University of Dortmund, Germany. It was originally proposed for micro-architecture design. One of the major advantages of MIMOLA is that the same description can be used for synthesis, simulation, test generation, and compilation. A tool chain including the MSSH hardware synthesizer, the MSSQ code generator, the MSST self-test program compiler, the MSSB functional simulator, and the MSSU RT-level simulator were developed based on the MIMOLA language [53]. MIMOLA has also been used by the RECORD [53] compiler.

MIMOLA description contains three parts: the algorithm to be compiled, the target processor model, and additional linkage and transformation rules. The software part (algorithm description) describes application programs in a PASCAL-like syntax. The processor model describes micro-architecture in the form of a component netlist. The linkage information is used by the compiler in order to locate important modules such as program counter and instruction memory. The following code segment specifies the program counter and instruction memory locations [53]:

```
LOCATION_FOR_PROGRAMCOUNTER PCReg;
LOCATION_FOR_INSTRUCTIONS IM[0..1023];
```

The algorithmic part of MIMOLA is an extension of PASCAL. Unlike other high level languages, it allows references to physical registers and memories. It also allows use of hardware components using procedure calls. For example, if the processor description contains a component named MAC, programmers can write the following code segment to use the multiply-accumulate operation performed by MAC:

```
res := MAC(x, y, z);
```

The processor is modeled as a net-list of component modules. MIMOLA permits modeling of arbitrary (programmable or non-programmable) hardware structures. Similar to VHDL, a number of predefined, primitive operators exists. The basic entities of MIMOLA hardware models are modules and connections. Each module is specified by its port interface and its behavior. The following example shows the description of a multi-functional ALU module [53]:

```
MODULE ALU
   (IN inp1, inp2: (31:0);
    OUT outp: (31:0);
    IN ctrl;
   )
CONBEGIN
       outp <- CASE ctrl OF
           0: inp1 + inp2 ;
           1: inp1 - inp2 ;
           END;
CONEND;
```

The CONBEGIN/CONEND construct includes a set of concurrent assignments. In the example a conditional assignment to output port *outp* is specified, which depends on the two-bit control input *ctrl*. The netlist structure is formed by connecting ports of module instances. For example, the following MIMOLA description connects two modules: *ALU* and accumulator *ACC*.

```
CONNECTIONS ALU.outp -> ACC.inp
            ACC.outp -> ALU.inp
```

The MSSQ code generator extracts instruction-set information from the module netlist. It uses two internal data structures: connection operation graph (COG) and instruction tree (I-tree). It is a very difficult task to extract the COG and I-trees even in the presence of linkage information due to the flexibility of an RT-level structural description. Extra constraints need to be imposed in order for the MSSQ code generator to work properly. The constraints limit the architecture scope of MSSQ to micro-programmable controllers, in which all control signals originate directly from the instruction word. The lack of explicit description of processor pipelines or resource conflicts may result in poor code quality for some classes of VLIW or deeply pipelined processors.

2.2.2 Behavioral ADLs

The difficulty of instruction-set extraction can be avoided by abstracting behavioral information from the structural details. Behavioral ADLs explicitly specify the instruction semantics and ignore detailed hardware structures. Typically, there is a one-to-one correspondence between behavioral ADLs and instruction-set reference manual. This section briefly describes two behavioral ADLs: nML [25] and ISDL [15].

nML

nML is an instruction-set oriented ADL proposed at Technical University of Berlin, Germany. nML has been used by code generators CBC [1] and CHESS [11], and instruction set simulators Sigh/Sim [13] and CHECKERS. Currently, CHESS/CHECKERS environment is used for automatic and efficient software compilation and instruction-set simulation [20].

nML developers recognized the fact that several instructions share common properties. The final nML description would be compact and simple if the common properties are exploited. Consequently, nML designers used a hierarchical scheme to describe instruction sets. The instructions are the topmost elements in the hierarchy. The intermediate elements of the hierarchy are partial instructions (PI). The relationship between elements can be established using two composition rules: AND-rule and OR-rule. The AND-rule groups several PIs into a larger PI and the OR-rule enumerates a set of alternatives for one PI. Therefore instruction definitions in nML can be in the form of an and-or tree. Each possible derivation of the tree corresponds to an actual instruction.

To achieve the goal of sharing instruction descriptions, the instruction set is enumerated by an attributed grammar [23]. Each element in the hierarchy has a few attributes. A non-leaf element's attribute values can be computed based on its children's attribute values. Attribute grammar is also adopted by other ADLs such as ISDL [15] and TDL [10]. The following nML description shows an example of instruction specification [25]:

```
op numeric_instruction(a:num_action, src:SRC, dst:DST)
action {
   temp_src = src;
   temp_dst = dst;
   a.action;
   dst = temp_dst;
}
op num_action = add | sub
op add()
action = {
   temp_dst = temp_dst + temp_src
}
```

The definition of *numeric_instruction* combines three partial instructions (PI) with the AND-rule: *num_action*, SRC, and DST. The first PI, *num_action*, uses OR-rule to describe the valid options for actions: *add* or *sub*. The number of all possible derivations of *numeric_instruction* is the product of the size of *num_action*, *SRC* and *DST*. The common behavior of all these options is defined in the *action* attribute of *numeric_instruction*. Each option for *num_action* should have its own action attribute defined as its specific behavior, which is referred by the *a.action* line. For example, the above code segment has action description for *add* operation. Object code image and assembly syntax can also be specified in the same hierarchical manner.

nML also captures the structural information used by instruction-set architecture (ISA). For example, storage units should be declared since they are visible to the instruction-set. nML supports three types of storages: RAM, register, and transitory storage. Transitory storage refers to machine states that are retained only for a limited number of cycles e.g., values on buses and latches. Computations have no delay in nML timing model - only storage units have delay. Instruction delay slots are modeled by introducing storage units as pipeline registers. The result of the computation is propagated through the registers in the behavior specification.

nML models constraints between operations by enumerating all valid combinations. The enumeration of valid cases can make nML descriptions lengthy. More complicated constraints, which often appear in DSPs with irregular instruction level parallelism (ILP) constraints or VLIW processors with multiple issue slots, are hard to model with nML. For example, nML cannot model the constraint that operation *I1* cannot directly follow operation *I0*. nML explicitly supports several addressing modes. However, it implicitly assumes an architecture model which restricts its generality. As a result it is hard to model multi-cycle or pipelined units and multi-word instructions explicitly. A good critique of nML is given in [26].

ISDL

Instruction Set Description Language (ISDL) was developed at MIT and used by the Aviv compiler [54] and GENSIM simulator generator [14]. The problem of constraint modeling is avoided by ISDL with explicit specification. ISDL is mainly targeted towards VLIW processors. Similar to nML, ISDL primarily describes the instruction-set of processor architectures. ISDL consists of mainly five sections: instruction word format, global definitions, storage resources, assembly syntax, and constraints. It also contains an optimization information section that can be used to provide certain architecture specific hints for the compiler to make better machine dependent code optimizations.

The instruction word format section defines fields of the instruction word. The instruction word is separated into multiple fields each containing one or more subfields. The global definition section describes four main types: tokens, non-terminals, split functions and macro definitions. Tokens are the primitive operands of instructions. For each token, assembly format and binary encoding information must be defined. An example token definition of a binary operand is:

```
Token X[0..1] X_R ival {yylval.ival = yytext[1] - '0';}
```

In this example, following the keyword *Token* is the assembly format of the operand. *X_R* is the symbolic name of the token used for reference. The *ival* is used to describe the value returned by the token. Finally, the last field describes the computation of the value. In this example, the assembly syntax allowed for the token *X_R* is *X0* or *X1*, and the values returned are 0 or 1 respectively. The value (last) field is to be used for behavioral definition and binary encoding assignment by non-terminals or instructions. Non-terminal is a mechanism provided to exploit commonalities among operations. The following code segment describes a non-terminal named *XYSRC*:

```
Non_Terminal ival XYSRC: X_D {$$ = 0;}  |
                         Y_D {$$ = Y_D + 1;};
```

The definition of *XYSRC* consists of the keyword *Non_Terminal*, the type of the returned value, a symbolic name as it appears in the assembly, and an action that describes the possible token or non-terminal combinations and the return value associated with each of them. In this example, *XYSRC* refers to tokens *X_D* and *Y_D* as its two options. The second field (*ival*) describes the returned value type. It returns 0 for *X_D* or incremented value for *Y_D*. Similar to nML, storage resources are the only structural information modeled by ISDL. The storage section lists all storage resources visible to the programmer. It lists the names and sizes of the memory, register files, and special registers. This information is used by the compiler to determine the available resources and how they should be used.

The assembly syntax section is divided into fields corresponding to the separate operations that can be performed in parallel within a single instruction. For each field, a list of alternative operations can be described. Each operation description consists of a name, a list of tokens or non-terminals as parameters, a set of commands that manipulate the bitfields, RTL description, timing details, and costs. RTL description captures the effect of the operation on the storage resources. Multiple costs are allowed including operation execution time, code size, and costs due to resource conflicts. The timing model of ISDL describes when the various effects of the operation take place (e.g., because of pipelining).

In contrast to nML, which enumerates all valid combinations, ISDL defines invalid combinations in the form of Boolean expressions. This often leads to a simple constraint specification. It also enables ISDL to capture irregular ILP constraints. The following example shows how to describe the constraint that instruction I1 cannot directly follow instruction I0. The "[1]" indicates a time shift of one instruction fetch for the I0 instruction. The '~' is a symbol for NOT and '&' is for logical AND.

```
~(I1 *) & ([1] I0 *, *)
```

ISDL provides the means for compact and hierarchical instruction set specification. However, it may not be possible to describe instruction sets with multiple encoding formats using simple tree-like instruction structure of ISDL.

2.2.3 Mixed ADLs

Mixed languages captures both structural and behavioral details of the architecture. This section briefly describes three mixed ADLs: HMDES, EXPRESSION, and LISA.

HMDES

Machine description language HMDES was developed at University of Illinois at Urbana-Champaign for the IMPACT research compiler [21]. C-like preprocessing capabilities such as file inclusion, macro expansion and conditional inclusion are supported in HMDES. An HMDES description is the input to the MDES machine description system of the Trimaran compiler infrastructure, which contains IMPACT as well as the Elcor research compiler from HP Labs. The description is first pre-processed, then optimized and translated to a low-level representation file. A machine database reads the low level files and supplies information for the compiler back end through a predefined query interface.

MDES captures both structure and behavior of target processors. Information is broken down into sections such as format, resource usage, latency, operation, and register. For example, the following code segment describes register and register file. It describes 64 registers. The register file describes the width of each register and other optional fields such as generic register type (virtual field), speculative, static and rotating registers. The value '1' implies speculative and '0' implies non-speculative.

```
SECTION Register {
  R0(); R1(); ... R63();
  'R[0]'(); ... 'R[63]'();
   ...
}

SECTION Register_ File {
  RF_i(width(32) virtual(i) speculative(1)
       static(R0...R63) rotating('R[0]'...'R[63]'));
  ...
}
```

MDES allows only a restricted retargetability of the cycle-accurate simulator to the HPL-PD processor family [33]. MDES permits description of memory systems, but limited to the traditional hierarchy, i.e., register files, caches, and main memory.

EXPRESSION

The above mixed ADLs require explicit description of Reservation Tables (RT). Processors that contain complex pipelines, large amounts of parallelism, and complex storage sub-systems, typically contain a large number of operations and resources (and hence RTs). Manual specification of RTs on a per-operation basis thus becomes cumbersome and error-prone. The manual specification of RTs (for each configuration) becomes impractical during rapid architectural exploration. The EXPRESSION ADL [3] describes a processor as a netlist of units and storages to automatically generate RTs based on the netlist [39]. Unlike MIMOLA, the netlist representation of EXPRESSION is coarse grain. It uses a higher level of abstraction similar to block-diagram level description in architecture manual.

EXPRESSION ADL was developed at University of California, Irvine. The ADL has been used by the retargetable compiler (EXPRESS [2]) and simulator (SIMPRESS [5]) generation framework. The framework also supports a graphical user interface (GUI) and can be used for design space exploration of programmable architectures consisting of processor cores, coprocessors and memories [18]. An EXPRESSION description is composed of two main sections: behavior (instruction-set), and structure. The behavior section has three subsections: operations, instruction, and operation mappings. Similarly, the structure section consists of three subsections: components, pipeline/data–transfer paths, and memory subsystem.

The operation subsection describes the instruction-set 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. A useful feature of EXPRESSION is operation group that groups similar operations together for the ease of later reference. For example, the following code segment shows an operation group (*alu_ops*) containing two ALU operations: *add* and *sub*.

```
(OP_GROUP alu_ops
   (OPCODE add
      (OPERANDS (SRC1 reg) (SRC2 reg/imm) (DEST reg))
      (BEHAVIOR DEST = SRC1 + SRC2)
   ...)
   (OPCODE sub
      (OPERANDS (SRC1 reg) (SRC2 reg/imm) (DEST reg))
      (BEHAVIOR DEST = SRC1 - SRC2)
   ...)
)
```

The instruction subsection captures the parallelism available in the architecture. Each instruction contains a list of slots (to be filled with operations), with each slot corresponding to a functional unit. The operation mapping subsection is used to specify the information needed by instruction selection and architecture-specific optimizations of the compiler. For example, it contains mapping between generic and target instructions.

The component subsection describes each RT-level component in the architecture. The components can be pipeline units, functional units, storage elements, ports, and connections. For multi-cycle or pipelined units, the timing behavior is also specified.

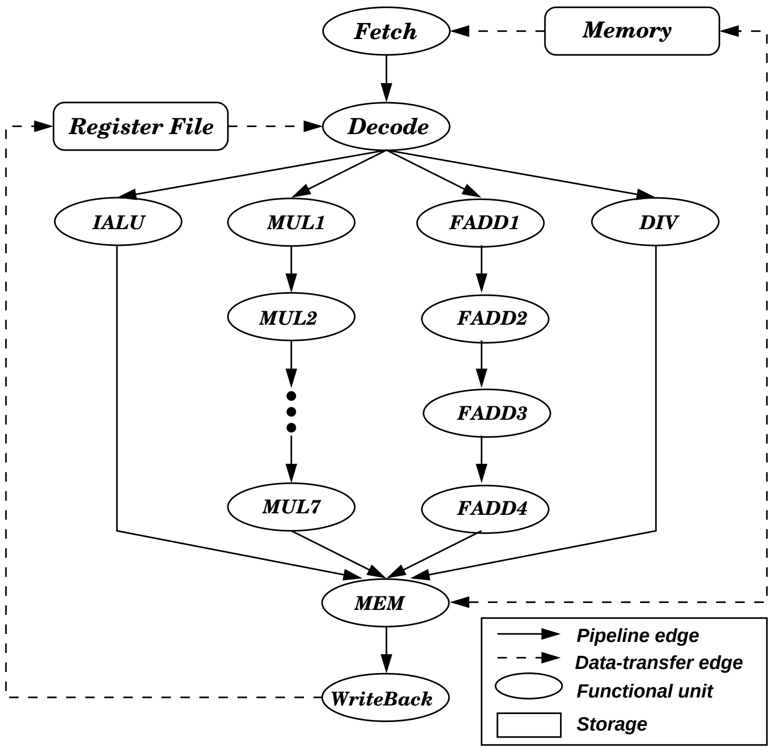

Figure 5. The DLX Architecture

The pipeline/data-transfer path subsection describes the netlist of the processor. The *pipeline path description* provides a mechanism to specify the units which comprise the pipeline stages, while the *data-transfer path 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 [39]. An example path declaration for the DLX architecture [22] (Figure 5) is shown below. It describes that the processor has five pipeline stages. It also describes that the *Execute* stage has four parallel paths. Finally, it describes each path e.g., it describes that the *FADD* path has four pipeline stages.

```
(PIPELINE Fetch Decode Execute MEM WriteBack)
(Execute (ALTERNATE IALU MULT FADD DIV))
(MULT (PIPELINE MUL1 MUL2 ... MUL7))
(FADD (PIPELINE FADD1 FADD2 FADD3 FADD4))
```

The memory subsection describes the types and attributes of various storage components (such as register files, SRAMs, DRAMs, and caches). The memory netlist information can be used to generate memory aware compilers and simulators [43]. Memory aware compilers can exploit the detailed information to hide the latency of the lengthy memory operations [40]. EXPRESSION captures the data path information in the processor. The control path is not explicitly modeled. The instruction model requires extension to capture inter-operation constraints such as sharing of common fields. Such constraints can be modeled by ISDL through cross-field encoding assignment.

LISA

LISA (Language for Instruction Set Architecture) [60] was developed at Aachen University of Technology, Germany with a simulator centric view. The language has been used to produce production quality simulators [55]. An important aspect of LISA language is its ability to capture control path explicitly. Explicit modeling of both datapath and control is necessary for cycle-accurate simulation. LISA has also been used to generate retargetable C compilers [27, 37]. LISA descriptions are composed of two types of declarations: resource and operation. The resource declarations cover hardware resources such as registers, pipelines, and memories. The pipeline model defines all possible pipeline paths that operations can go through. An example pipeline description for the architecture shown in Figure 5 is as follows:

```
PIPELINE int = {Fetch; Decode; IALU; MEM; WriteBack}
PIPELINE flt = {Fetch; Decode; FADD1; FADD2;
                FADD3; FADD4; MEM; WriteBack}
PIPELINE mul = {Fetch; Decode; MUL1; MUL2; MUL3; MUL4;
                MUL5; MUL6; MUL7; MEM; WriteBack}
PIPELINE div = {Fetch; Decode; DIV; MEM; WriteBack}
```

Operations are the basic objects in LISA. They represent the designer's view of the behavior, the structure, and the instruction set of the programmable architecture. Operation definitions capture the description of different properties of the system such as operation behavior, instruction set information, and timing. These operation attributes are defined in several sections. LISA exploits the commonality of similar operations by grouping them into one. The following code segment describes the decoding behavior of two immediate-type (i_type) operations (ADDI and SUBI) in the DLX *Decode* stage. The complete behavior of an operation can be obtained by combining its behavior definitions in all the pipeline stages.

```
OPERATION i_type IN pipe_int.Decode {
    DECLARE {
        GROUP opcode={ADDI || SUBI}
        GROUP rs1, rd = {fix_register};
    }
    CODING {opcode rs1 rd immediate}
    SYNTAX {opcode rd ‘‘,’’ rs1 ‘‘,’’ immediate}
    BEHAVIOR { reg_a = rs1; imm = immediate; cond = 0;
    }
    ACTIVATION {opcode, writeback}
}
```

A language similar to LISA is RADL. RADL [9] was developed at Rockwell, Inc. as an extension of the LISA approach that focuses on explicit support of detailed pipeline behavior to enable generation of production quality cycle-accurate and phase-accurate simulators.

2.2.4 Partial ADLs

The ADLs discussed so far captures complete description of the processor's structure, behavior or both. There are many description languages such as AIDL [57] that captures partial information of the architecture needed to perform specific task. AIDL is an ADL developed at University of Tsukuba for design of high-performance superscalar processors [57]. It seems that AIDL does not aim at datapath optimization but aims at validation of the pipeline behavior such as data-forwarding and out-of-order completion. In AIDL, timing behavior of pipeline is described using interval temporal logic. AIDL does not support software toolkit generation. However, AIDL descriptions can be simulated using the AIDL simulator.

3 ADL-driven Methodologies

This section describes the ADL-driven methodologies used for processor development. It presents the following three methodologies that are used in academic research as well as industry:

- Software toolkit generation and exploration
- Generation of hardware implementation
- Top-down validation

3.1 Software Toolkit Generation and Exploration

Embedded systems present a tremendous opportunity to customize designs by exploiting the application behavior. Rapid exploration and evaluation of candidate architectures are necessary due to time-to-market pressure and short product lifetimes. ADLs are used to specify processor and memory architectures and generate software toolkit including compiler, simulator, assembler, profiler and debugger. Figure 6 shows a traditional ADL-based design space exploration flow. The application programs are compiled and simulated, and the feedback is used to modify the ADL specification with the goal of finding the best possible architecture for the given set of application programs under various design constraints such as area, power, and performance.

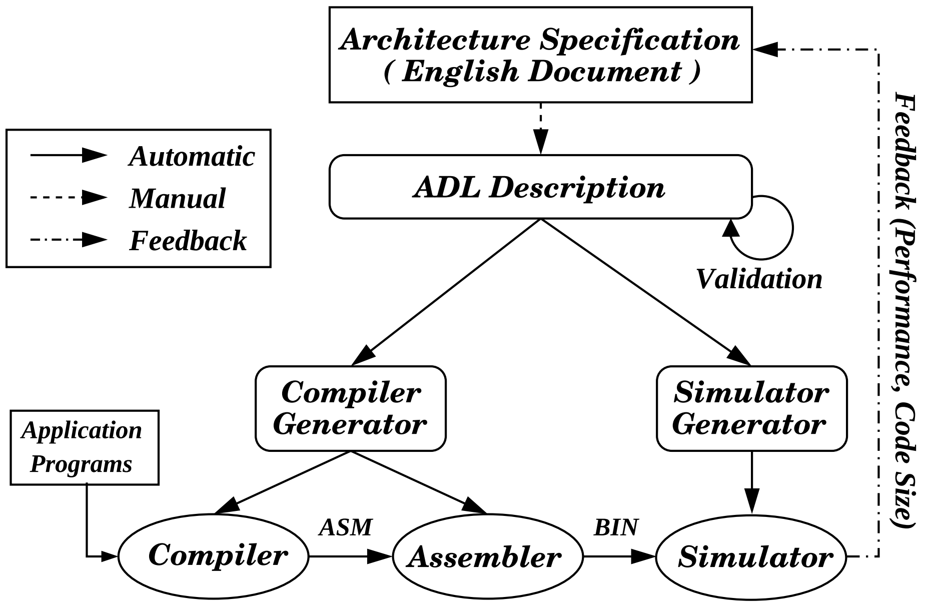

Figure 6. ADL driven Design Space Exploration

An extensive body of recent work addresses ADL driven software toolkit generation and design space exploration of processor-based embedded systems, in both academia: ISDL [15], Valen-C [4], MIMOLA [53], LISA [60], nML [25], Sim-nML [59], EXPRESSION [3], and industry: ARC [8], Axys [17], RADL [9], Target [20], Tensilica [58], MDES [33].

One of the main purposes of an ADL is to support automatic generation of a high-quality software toolkit including at least an ILP (instruction level parallelism) compiler and a cycle-accurate simulator. However, such tools require detailed information about the processor, typically in a form that is not concise and easily specifiable. Therefore, it becomes necessary to develop procedures to automatically generate such tool-specific information from the ADL specification. For example, reservation tables (RTs) are used in many ILP compiler to describe resource conflicts. However, manual description of RTs on a per-instruction basis is cumbersome and error-prone. Instead, it is easier to specify the pipeline and datapath resources in an abstract manner, and generate RTs on a per-instruction basis [39]. This section describes some of the challenges in automatic generation of software tools (focusing on compilers and simulators) and survey some of the approaches adopted by current tools.

3.1.1 Compilers

Traditionally, software for embedded systems was hand-tuned in assembly. With increasing complexity of embedded systems, it is no longer practical to develop software in assembly language or to optimize it manually except for critical sections of the code. Compilers which produce optimized machine specific code from a program specified in a high-level language (HLL) such as C/C++ and Java are necessary in order to produce efficient software within the time budget. Compilers for embedded systems have been the focus of several research efforts recently [41].

The compilation process can be broadly broken into two steps: analysis and synthesis. During analysis, the program (in HLL) is converted into an intermediate representation (IR) that contains all the desired information such as control and data dependences. During synthesis, the IR is transformed and optimized in order to generate efficient target specific code. The synthesis step is more complex and typically includes the following phases: instruction selection, scheduling, resource allocation, code optimizations/transformations, and code generation. The effectiveness of each phase depends on the algorithms chosen and the target architecture. A further problem during the synthesis step is that the optimal ordering between these phases is highly dependent on the target architecture and the application program. As a result, traditionally, compilers have been painstakingly hand-tuned to a particular architecture (or architecture class) and application domain(s). However, stringent time-to-market constraints for SOC designs no longer make it feasible to manually generate compilers tuned to particular architectures. Automatic generation of an efficient compiler from an abstract description of the processor model becomes essential.

A promising approach to automatic compiler generation is the "retargetable compiler" approach. A compiler is classified as retargetable if it can be adapted to generate code for different target processors with significant reuse of the compiler source code. Retargetability is typically achieved by providing target machine information (in an ADL) as input to the compiler along with the program corresponding to the application. The complexity in retargeting the compiler depends on the range of target processors it supports and also on its optimizing capability. Due to the growing amount of Instruction Level Parallelism (ILP) features in modern processor architectures, the difference in quality of code generated by a naive code conversion process and an optimizing ILP compiler can be enormous. Recent approaches on retargetable compilation have focused on developing optimizations/transformations that are "retargetable" and capturing the machine specific information needed by such optimizations in the ADL. The retargetable compilers can be classified into three broad categories, based on the type of the machine model accepted as input.

Architecture template based

Such compilers assume a limited architecture template which is parameterizable for customization. The most common parameters include operation latencies, number of functional units, number of registers, etc. Architecture template based compilers have the advantage that both optimizations and the phase ordering between them can be manually tuned to produce highly efficient code for the limited architecture space. Examples of such compilers include the Valen-C compiler [4] and the GNU-based C/C++ compiler from Tensilica Inc. [58]. The Tensilica GNU-based C/C++ compiler is geared towards the Xtensa parameterizable processor architecture. One important feature of this system is the ability to add new instructions (described through an Instruction Extension Language) and automatically generate software tools tuned to the new instruction-set.

Explicit behavioral information based

Most compilers require a specification of the behavior in order to retarget their transformations (e.g., instruction selection requires a description of the semantics of each operation). Explicit behavioral information based retargetable compilers require full information about the instruction-set as well as explicit resource conflict information. Examples include the AVIV [54] compiler using ISDL, CHESS [11] using nML, and Elcor [33] using MDES. The AVIV retargetable code generator produces machine code, optimized for minimal size, for target processors with different instruction-set. It solves the phase ordering problem by performing a heuristic branch-and-bound step that performs resource allocation/assignment, operation grouping, and scheduling concurrently. CHESS is a retargetable code generation environment for fixed-point DSP processors. CHESS performs instruction selection, register allocation, and scheduling as separate phases (in that order). Elcor is a retargetable compilation environment for VLIW architectures with speculative execution. It implements a software pipelining algorithm (modulo scheduling) and register allocation for static and rotating register files.

Behavioral information generation based

Recognizing that the architecture information needed by the compiler is not always in a form that may be well suited for other tools (such as synthesis) or does not permit concise specification, some research has focussed on extraction of such information from a more amenable specification. Examples include the MSSQ and RECORD compiler using MIMOLA [53], retargetable C compiler based on LISA [27], and the EXPRESS compiler using EXPRESSION [3]. MSSQ translates Pascal-like high-level language (HLL) into microcode for micro-programmable controllers, while RECORD translates code written in a DSP-specific programming language, called data flow language (DFL), into machine code for the target DSP. The retargetable C compiler generation using LISA is based on reuse of a powerful C compiler platform with many built-in code optimizations and generation of mapping rules for code selection using the instruction semantics information [27]. The EXPRESS compiler tries to bridge the gap between explicit specification of all information (e.g., AVIV) and implicit specification requiring extraction of instruction-set (e.g., RECORD), by having a mixed behavioral/structural view of the processor.

3.1.2 Simulators

Simulators are critical components of the exploration and software design toolkit for the system designer. They can be used to perform diverse tasks such as verifying the functionality and/or timing behavior of the system (including hardware and software), and generating quantitative measurements (e.g. power consumption) which can be used to aid the design process.

Simulation of the processor system can be performed at various abstraction levels. At the highest level of abstraction, a functional simulation of the processor can be performed by modeling only the instruction-set (IS). Such simulators are termed instruction-set simulators (ISS) or instruction-level simulators (ILS). At lower-levels of abstraction are the cycle-accurate and phase-accurate simulation models that yield more detailed timing information. Simulators can be further classified based on whether they provide bit-accurate models, pin-accurate models, exact pipeline models, and structural models of the processor.

Typically, simulators at higher levels of abstraction (e.g. ISS, ILS) are faster but gather less information as compared to those at lower levels of abstraction (e.g., cycle-accurate, phase-accurate). Retargetability (i.e. ability to simulate a wide variety of target processors) is especially important in the arena of embedded SOC design with emphasis on the DSE and co-development of hardware and software. Simulators with limited retargetability are very fast but may not be useful in all aspects of the design process. Such simulators typically incorporate a fixed architecture template and allow only limited retargetability in the form of parameters such as number of registers and ALUs. Examples of such simulators are numerous in the industry and include the HPL-PD [33] simulator using the MDES ADL. The model of simulation adopted has significant impact on the simulation speed and flexibility of the simulator. Based on the simulation model, simulators can be classified into three types: interpretive, compiled, and mixed.

Interpretation based

Such simulators are based on an interpretive model of the processors instruction-set. Interpretive simulators store the state of the target processor in host memory. It then follows a fetch, decode, and execute model: instructions are fetched from memory, decoded and then executed in serial order. Advantages of this model include ease of implementation, flexibility and the ability to collect varied processor state information. However, it suffers from significant performance degradation as compared to the other approaches primarily due to the tremendous overhead in fetching, decoding and dispatching instructions. Almost all commercially available simulators are interpretive. Examples of research interpretive retargetable simulators include SIMPRESS [5] using EXPRESSION, and GENSIM/XSIM [14] using ISDL.

Compilation based

Compilation based approaches reduce the runtime overhead by translating each target instruction into a series of host machine instructions which manipulate the simulated machine state. Such translation can be done either at compile time (static compiled simulation) where the fetch-decode-dispatch overhead is completely eliminated, or at load time (dynamic compiled simulation) which amortizes the overhead over repeated execution of code. Simulators based on the static compilation model are presented by Zhu et al. [24] and Pees et al. [55]. Examples of dynamic compiled code simulators include the Shade simulator [52], and the Embra simulator [12].

Interpretive+Compiled

Traditional interpretive simulation is flexible but slow. Instruction decoding is a time consuming process in a software simulation. Compiled simulation performs compile time decoding of application programs to improve the simulation performance. However, all compiled simulators rely on the assumption that the complete program code is known before the simulation starts and is further more run-time static. Due to the restrictiveness of the compiled technique, interpretive simulators are typically used in embedded systems design flow. Two recently proposed simulation techniques (JIT-CCS [6] and IS-CS [32]) combines the flexibility of interpretive simulation with the speed of the compiled simulation.

The *just-in-time cache compiled simulation* (JIT-CCS) technique compiles an instruction during run-time, *just-in-time* before the instruction is going to be executed. Subsequently, the extracted information is stored in a simulation cache for direct reuse in a repeated execution of the program address. The simulator recognizes if the program code of a previously executed address has changed and initiates a re-compilation. The *instruction set compiled simulation* (IS-CS) technique performs time-consuming instruction decoding during compile time. In case, an instruction is modified at run-time, the instruction is re-decoded prior to execution. It also uses an *instruction abstraction* technique to generate aggressively optimized decoded instructions that further improves simulation performance [31, 32].

3.2 Generation of Hardware Implementation

Recent approaches on ADL-based software toolkit generation enables performance driven exploration. The simulator produces profiling data and thus may answer questions concerning the instruction set, the performance of an algorithm and the required size of memory and registers. However, the required silicon area, clock frequency, and power consumption can only be determined in conjunction with a synthesizable HDL model. There are two major approaches in the literature for synthesizable HDL generation. The first one is a parameterized processor core based approach. These cores are bound to a single processor template whose architecture and tools can be modified to a certain degree. The second approach is based on processor specification languages.

Processor template based

Examples of processor template based approaches are Xtensa [58], Jazz [19], and PEAS [28]. Xtensa [58] is a scalable RISC processor core. Configuration options include the width of the register set, caches, and memories. New functional units and instructions can be added using the Tensilica Instruction Language (TIE). A synthesizable hardware model along with software toolkit can be generated for this class of architectures. Improv's Jazz [19] processor is supported by a flexible design methodology to customize the computational resources and instruction set of the processor. It allows modifications of data width, number of registers, depth of hardware task queue, and addition of custom functionality in Verilog. PEAS [28] is a GUI based hardware/software codesign framework. It generates HDL code along with software toolkit. It has support for several architecture types and a library of configurable resources.

Specification language based

Figure 7 shows a typical framework of processor description language driven HDL generation. Structure-centric ADLs such as MIMOLA are suitable for hardware generation. Some of the behavioral languages (such as ISDL and nML) are also used for hardware generation. For example, the HDL generator HGEN [14] uses ISDL description, and the synthesis tool GO [20] is based on nML. Itoh et al. [29] have proposed a micro-operation description based synthesizable HDL generation. It can handle simple processor models with no hardware interlock mechanism or multi-cycle operations.

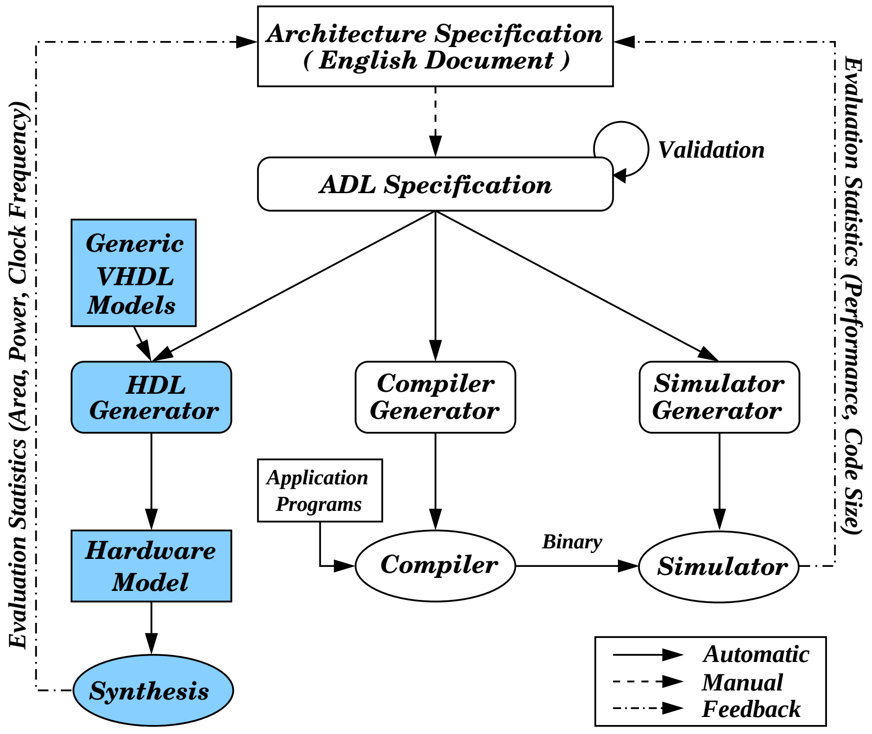

Figure 7. ADL-driven Implementation Generation and Exploration

Mixed languages such as LISA and EXPRESSION capture both structure and behavior of the processor. The synthesizable HDL generation approach based on LISA language [35] produces an HDL model of the architecture. The designer has the choice to generate a VHDL, Verilog or SystemC representation of the target architecture [35]. The HDL generation methodology presented by Mishra et al. [42] combines the advantages of the processor template based environments and the language based specifications using EXPRESSION ADL.

3.3 Top-Down Validation

Validation of microprocessors is one of the most complex and important tasks in the current System-on-Chip (SOC) design methodology. Figure 8 shows a traditional architecture validation flow. The architect prepares an informal specification of the microprocessor in the form of an English document. The logic designer implements the modules in the register-transfer level (RTL). The *RTL design* is validated using a combination of simulation techniques and formal methods. One of the most important problems in today's processor design validation is the lack of a golden reference model that can be used for verifying the design at different levels of abstraction. Thus, many existing validation techniques employ a *bottom-up approach* to pipeline verification, where the functionality of an existing pipelined processor is, in essence, reverse-engineered from its RT-level implementation.

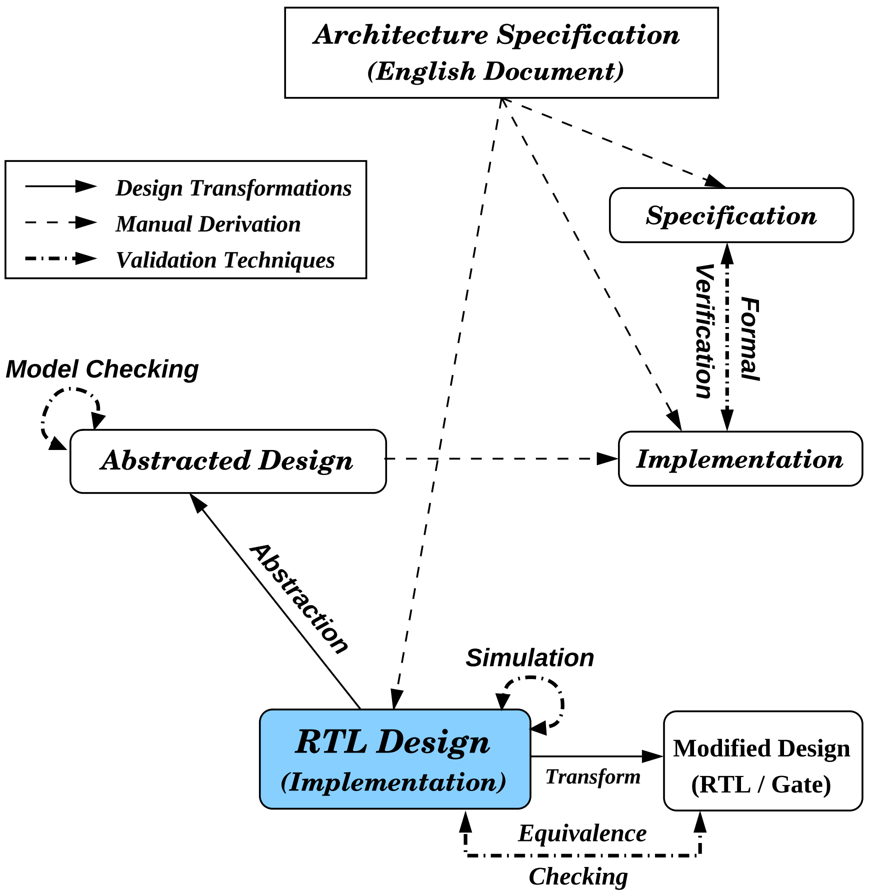

Figure 8. Traditional Bottom-Up Validation Flow

Mishra et al. [51] have presented an ADL-driven validation technique that is complementary to these bottom-up approaches. It leverages the system architects knowledge about the behavior of the programmable architectures through ADL constructs, thereby allowing a powerful *top-down approach* to microprocessor validation. Figure 9 shows an ADL-driven top-down validation methodology. This methodology has two important steps: validation of ADL specification, and specification-driven validation of programmable architectures.

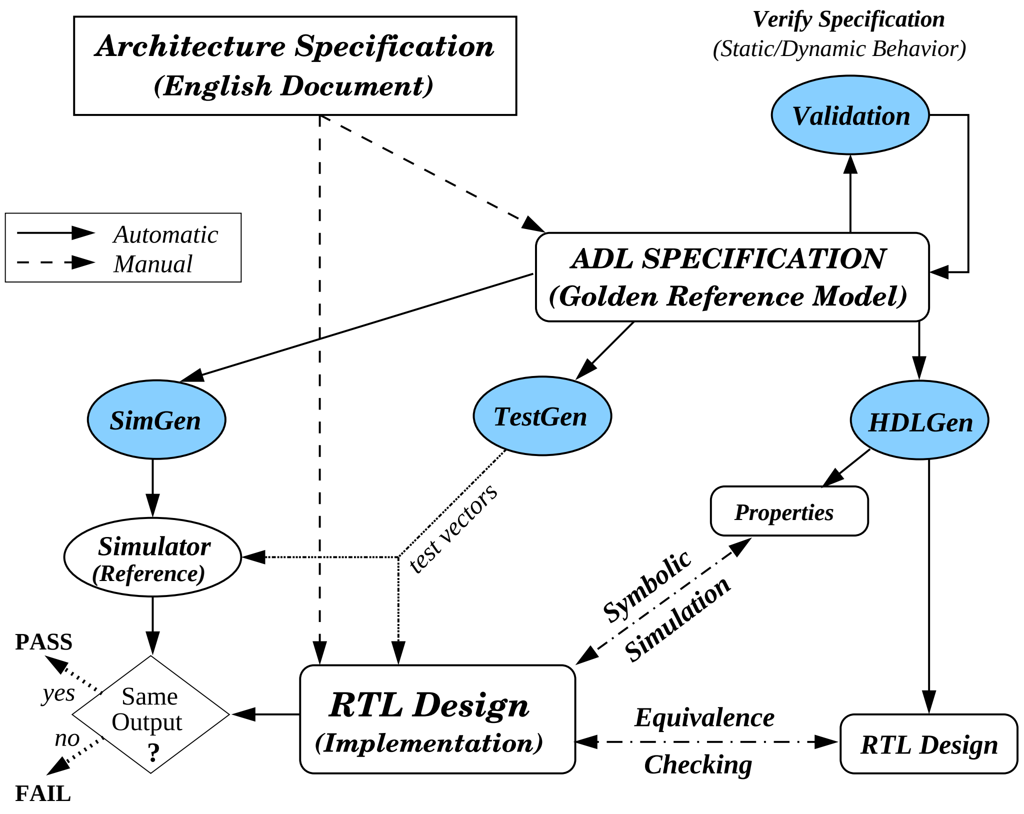

Figure 9. Top-Down Validation Flow

Validation of ADL Specification

It is important to verify the ADL specification to ensure the correctness of the architecture specified and the generated software toolkit. Both static and dynamic behavior need to be verified to ensure that the specified architecture is well-formed. The static behavior can be validated by analyzing several static properties such as, connectedness, false pipeline and data-transfer paths and completeness using a graph based model of the pipelined architecture [44]. The dynamic behavior can be validated by analyzing the instruction flow in the pipeline using a Finite State Machine (FSM) based model to verify several important architectural properties such as determinism and in-order execution in the presence of hazards and multiple exceptions [48].

Specification-driven Validation

The validated ADL specification can be used as a golden reference model for top-down validation of programmable architectures. The top-down validation approach has been demonstrated in two directions: functional test program generation, and design validation using a combination of equivalence checking and symbolic simulation.

Test generation for functional validation of processors has been demonstrated using MIMOLA [53], EXPRESSION [45], and nML [20]. A model checking based approach is used to automatically generate functional test programs from the processor specification using EXPRESSION ADL [45]. It generates graph model of the pipelined processor from the ADL specification. The functional test programs are generated based on the coverage of the pipeline behavior. ADL-driven design validation using equivalence checking has been demonstrated using EXPRESSION ADL [49]. This approach combines ADL-driven hardware generation and validation. The generated hardware model (RTL) is used as a reference model to verify the hand-written implementation (*RTL design*) of the processor. To verify that the implementation satisfies certain properties, the framework generates the intended properties. These properties are applied using symbolic simulation [49].

4 Conclusions

Design and verification of today's complex processors requires the use of automated tools and techniques. ADLs have been successfully used in academic research as well as industry for processor development. The processor is modeled using an ADL. The ADL specification is used to generate software tools including compiler and simulator to enable early design space exploration. The ADL specification is also used to perform other design automation tasks including hardware generation and functional verification.

The early ADLs were either structure-oriented (MIMOLA, UDL/I), or behavior-oriented (nML, ISDL). As a result, each class of ADLs are suitable for specific tasks. For example, structure-oriented ADLs are suitable for hardware synthesis, and unfit for compiler generation. Similarly, behavior-oriented ADLs are appropriate for generating compiler and simulator for instruction-set architectures, and unsuited for generating cycle-accurate simulator or hardware implementation of the architecture. The later ADLs (LISA and EXPRESSION) adopted the mixed approach where the language captures both structure and behavior of the architecture.

The existing ADLs are getting modified with the new features and methodologies to perform software toolkit generation, hardware generation, instruction-set synthesis, and test generation for validation of architectures. For example, nML is extended by Target Compiler Technologies to perform hardware synthesis and test generation [20]. Similarly, LISA language has been used for hardware generation [35], instruction encoding synthesis [7], and JTAG interface generation [36]. Likewise, EXPRESSION has been used for hardware generation [42], instruction-set synthesis [38], test generation [47], and specification validation [51].

ADLs designed for a specific domain (such as DSP or VLIW) or for a specific purpose (such as simulation or compilation) can be compact and it is possible to automatically generate efficient (in terms of area, power and performance) tools/hardwares. However, it is difficult to design an ADL for a wide variety of architectures to perform different tasks using the same specification. Generic ADLs require the support of powerful methodologies to generate high quality results compared to domain-specific/task-specific ADLs.

In the future, the existing ADLs will go through changes in two dimensions. First, ADLs will specify not only processor, memory and co-processor architectures but also other components of the system-on-chip architectures including peripherals and external interfaces. Second, ADLs will be used for software toolkit generation, hardware synthesis, test generation, instruction-set synthesis, and validation of microprocessors. Furthermore, multiprocessor SOCs will be captured and various attendant tasks will be addressed. The tasks include support for formal analysis, generation of real-time operating systems (RTOS), exploration of communication architectures, and support for interface synthesis. The emerging ADLs will have these features.

References

[1] A. Fauth and A. Knoll. Automatic generation of DSP program development tools. In *Proceedings of Int'l Conf. Acoustics, Speech and Signal Processing (ICASSP)*, pages 457–460, 1993.

[2] A. Halambi and A. Shrivastava and N. Dutt and A. Nicolau. A customizable compiler framework for embedded systems. In *Proceedings of Software and Compilers for Embedded Systems (SCOPES)*, 2001.

[3] A. Halambi and P. Grun and V. Ganesh and A. Khare and N. Dutt and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 485–490, 1999.

[4] A. Inoue and H. Tomiyama and F. Eko and H. Kanbara and H. Yasuura. A programming language for processor based embedded systems. In *Proceedings of Asia Pacific Conference on Hardware Description Languages (APCHDL)*, pages 89–94, 1998.

[5] A. Khare and N. Savoiu and A. Halambi and P. Grun and N. Dutt and A. Nicolau. V-SAT: A visual specification and analysis tool for system-on-chip exploration. In *Proceedings of EUROMICRO Conference*, pages 1196–1203, 1999.

[6] A. Nohl and G. Braun and O. Schliebusch and R. Leupers and H. Meyr and A. Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *Proceedings of Design Automation Conference (DAC)*, pages 22–27, 2002.

[7] A. Nohl and V. Greive and G. Braun and A. Hoffmann and R. Leupers and O. Schliebusch and H. Meyr. Instruction encoding synthesis for architecture exploration using hierarchical processor models. In *Proceedings of Design Automation Conference (DAC)*, pages 262–267, 2003.

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

[9] C. Siska. A processor description language supporting retargetable multi-pipeline DSP program development tools. In *Proceedings of International Symposium on System Synthesis (ISSS)*, pages 31–36, 1998.

[10] D. Kastner. TDL: A hardware and assembly description languages. Technical Report TDL 1.4, Saarland University, Germany, 2000.

[11] D. Lanneer and J. Praet and A. Kifli and K. Schoofs and W. Geurts and F. Thoen and G. Goossens. CHESS: Retargetable code generation for embedded DSP processors. In *Code Generation for Embedded Processors.*, pages 85–102. Kluwer Academic Publishers, 1995.

[12] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

[13] F. Lohr and A. Fauth and M. Freericks. Sigh/sim: An environment for retargetable instruction set simulation. Technical Report 1993/43, Dept. Computer Science, Tech. Univ. Berlin, Germany, 1993.

[14] G. Hadjiyiannis and P. Russo and S. Devadas. A methodology for accurate performance evaluation in architecture exploration. In *Proceedings of Design Automation Conference (DAC)*, pages 927–932, 1999.

[15] G. Hadjiyiannis and S. Hanono and S. Devadas. ISDL: An instruction set description language for retargetability. In *Proceedings of Design Automation Conference (DAC)*, pages 299–302, 1997.

[16] H. Tomiyama and A. Halambi and P. Grun and N. Dutt and A. Nicolau. Architecture description languages for systems-on-chip design. In *Proceedings of Asia Pacific Conference on Chip Design Language*, pages 109–116, 1999.

[17] http://www.axysdesign.com. *Axys Design Automation.*

[18] http://www.ics.uci.edu/˜express. *Exploration framework using EXPRESSION.*

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

[20] http://www.retarget.com. *Target Compiler Technologies.*

[21] J. Gyllenhaal and B. Rau and W. Hwu. HMDES version 2.0 specification. Technical Report IMPACT-96-3, IMPACT Research Group, Univ. of Illinois, Urbana. IL, 1996.

[22] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc, San Mateo, CA, 1990.

[23] J. Paakki. Attribute grammar paradigms - a high level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–256, June 1995.

[24] J. Zhu and D. Gajski. A retargetable, ultra-fast, instruction set simulator. In *Proceedings of Design Automation and Test in Europe (DATE)*, 1999.

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

[26] M. Hartoog and J. Rowson and P. Reddy and S. Desai and D. Dunlop and E. Harcourt and N. Khullar. Generation of software tools from processor descriptions for hardware/software codesign. In *Proceedings of Design Automation Conference (DAC)*, pages 303–306, 1997.

[27] M. Hohenauer and H. Scharwaechter and K. Karuri and O. Wahlen and T. Kogel and R. Leupers and G. Ascheid and H. Meyr and G. Braun and H. Someren. A methodology and tool suite for c compiler generation from ADL processor models. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 1276–1283, 2004.

[28] M. Itoh and S. Higaki and Y. Takeuchi and A. Kitajima and M. Imai and J. Sato and A. Shiomi. PEAS-III: An ASIP design environment. In *Proceedings of International Conference on Computer Design (ICCD)*, 2000.

[29] M. Itoh and Y. Takeuchi and M. Imai and A. Shiomi. Synthesizable HDL generation for pipelined processors from a micro-operation description. *IEICE Trans. Fundamentals*, E00-A(3), March 2000.

[30] M. R. Barbacci. Instruction set processor specifications (ISPS): The notation and its applications. *IEEE Transactions on Computers*, C-30(1):24–40, Jan 1981.

[31] M. Reshadi and N. Bansal and P. Mishra and N. Dutt. An efficient retargetable framework for instruction-set simulation. In *Proceedings of International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 13–18, 2003.

[32] M. Reshadi and P. Mishra and N. Dutt. Instruction set compiled simulation: A technique for fast and flexible instruction set simulation. In *Proceedings of Design Automation Conference (DAC)*, pages 758–763, 2003.

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

[34] N. Medvidovic and R. Taylor. A framework for classifying and comparing architecture description languages. In M. J. and H. Schauer, editor, *Proceedings of European Software Engineering Conference (ESEC)*, pages 60–76. Springer–Verlag, 1997.

[35] O. Schliebusch and A. Chattopadhyay and M. Steinert and G. Braun and A. Nohl and R. Leupers and G. Ascheid and H. Meyr. RTL processor synthesis for architecture exploration and implementation. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 156–160, 2004.

[36] O. Schliebusch and D. Kammler and A. Chattopadhyay and R. Leupers and G. Ascheid and H. Meyr. Automatic generation of JTAG interface and debug mechanism for ASIPs. In *GSPx*, 2004.

[37] O. Wahlen and M. Hohenauer and R. Leupers and H. Meyr. Instruction scheduler generation for retragetable compilation. *IEEE Design & Test of Computers*, 20(1):34–41, Jan-Feb 2003.

[38] P. Biswas and N. Dutt. Reducing code size for heterogeneous-connectivity-based VLIW DSPs through synthesis of instruction set extensions. In *Proceedings of Compilers, Architectures, Synthesis for Embedded Systems (CASES)*, pages 104–112, 2003.

[39] P. Grun and A. Halambi and N. Dutt and A. Nicolau. RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions. In *Proceedings of International Symposium on System Synthesis (ISSS)*, pages 44–50, 1999.

[40] P. Grun and N. Dutt and A. Nicolau. Memory aware compilation through accurate timing extraction. In *Proceedings of Design Automation Conference (DAC)*, pages 316–321, 2000.

[41] P. Marwedel and G. Goossens. Code generation for embedded processors. Kluwer Academic Publishers, 1995.

[42] P. Mishra and A. Kejariwal and N. Dutt. Synthesis-driven exploration of pipelined embedded processors. In *Proceedings of International Conference on VLSI Design*, 2004.

[43] P. Mishra and M. Mamidipaka and N. Dutt. Processor-memory co-exploration using an architecture description language. *To appear, ACM Transactions on Embedded Computing Systems (TECS)*, 3(1):140–162, 2004.

[44] P. Mishra and N. Dutt. Automatic modeling and validation of pipeline specifications. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(1):114–139, 2004.

[45] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 182–187, 2004.

[46] P. Mishra and N. Dutt. Architecture description languages for programmable embedded systems. *IEE Proceedings on Computers and Digital Techniques*, 2005.

[47] P. Mishra and N. Dutt. Functional coverage driven test generation for validation of pipelined processors. In *Proceedings of Design Automation and Test in Europe (DATE)*, 2005.

[48] P. Mishra and N. Dutt and H. Tomiyama. Towards automatic validation of dynamic behavior in pipelined processor specifications. *Kluwer Design Automation for Embedded Systems(DAES)*, 8(2-3):249–265, June-September 2003.

[49] P. Mishra and N. Dutt and N. Krishnamurthy and M. Abadir. A top-down methodology for validation of microprocessors. *IEEE Design & Test of Computers*, 21(2):122–131, 2004.

[50] Paul C. Clements. A survey of architecture description languages. In *Proceedings of International Workshop on Software Specification and Design (IWSSD)*, pages 16–25, 1996.

[51] Prabhat Mishra. *Specification-driven Validation of Programmable Embedded Systems.* PhD thesis, University of California, Irvine, March 2004.

[52] R. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.

[53] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1):75–108, 1998.

[54] S. Hanono and S. Devadas. Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator. In *Proceedings of Design Automation Conference (DAC)*, pages 510–515, 1998.

[55] S. Pees and A. Hoffmann and H. Meyr. Retargetable compiled simulation of embedded processors using a machine description language. *ACM Transactions on Design Automation of Electronic Systems*, 5(4):815–834, Oct. 2000.

[56] S. Wang and S. Malik. Synthesizing operating system based device drivers in embedded systems. In *Proceedings of International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 37–44, 2003.

[57] T. Morimoto and K. Yamazaki and H. Nakamura and T. Boku and K. Nakazawa. Superscalar processor design with hardware description language aidl. In *Proceedings of Asia Pacific Conference on Hardware Description Languages (APCHDL)*, 1994.

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

[59] V. Rajesh and Rajat Moona. Processor modeling for hardware software codesign. In *Proceedings of International Conference on VLSI Design*, pages 132–137, 1999.

[60] V. Zivojnovic and S. Pees and H. Meyr. LISA - machine description language and generic machine model for HW/SW co-design. In *IEEE Workshop on VLSI Signal Processing*, pages 127–136, 1996.

[61] W. Qin and S. Malik. *Architecture Description Languages for Retargetable Compilation, in The Compiler Design Handbook: Optimizations & Machine Code Generation.* CRC Press, 2002.