# Architecture Description Languages

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*

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

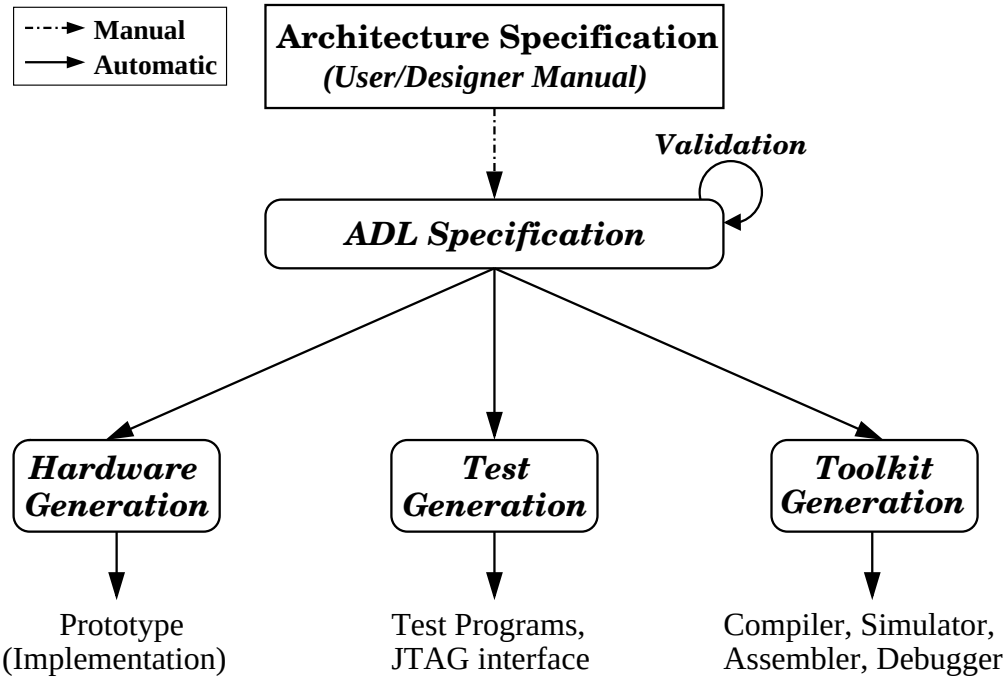


**Figure 1. ADL-driven design automation of embedded processors**

Architecture Description Languages (ADL) enable design automation of embedded processors as as shown in Figure 1. The ADL specification is used to generate various executable models including simulator, compiler and hardware implementation. The generated models enable various design automation tasks including exploration, simulation, compilation, synthesis, test generation, and validation. Chapter 5 describes retargetable software tools for embedded processors. This chapter reviews the existing ADLs in terms of their capabilities in capturing a wide variety of embedded processors available today. Existing ADLs can be classified 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. Existing ADLs can be classified into various content-based categories such as structural, behavioral and mixed ADLs. Similarly, contemporary ADLs can be classified into various objective-oriented categories such as simulation-oriented, synthesis-oriented, test-oriented, compilation-oriented, validation-oriented, and so on. This chapter is organized as follows. Section 1 describes how ADLs differ from other modeling languages. Section 2 surveys the contemporary ADLs. Finally, Section 3 concludes this chapter with a discussion on expected features of future ADLs.

## 1 ADLs and other Languages

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 [31]. 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 [19] were used for simulation, evaluation, and synthesis of computers and other digital systems. This chapter surveys contemporary hardware ADLs.

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? Specifications widely in use today are still written informally in natural languages such as 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. An ADL specification should have formal (unambiguous) semantics as well as easy correlation with the architecture manual.

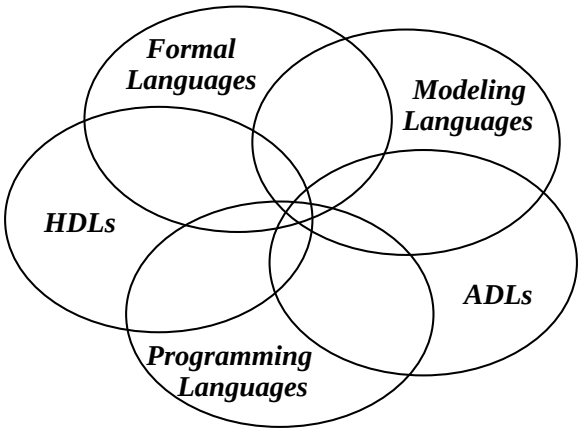


**Figure 2. ADLs versus non-ADLs**

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 in 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 embedded processors.

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 2. 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.

## 2 Survey of Contemporary ADLs

This section briefly surveys some of the contemporary ADLs in the context of designing customizable and configurable embedded processors. There are many comprehensive ADL surveys available in the literature including ADLs for retargetable compilation [37] and SOC design [10]. Figure 3 shows the classification of 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 four categories based on the objective: simulation-oriented, synthesis-oriented, compilation-oriented and validation-oriented. It is not always possible to establish a one-to-one correspondence between content-based and objective-based classification.

### 2.1 Content-oriented Classification of ADLs

ADLs can be classified into four categories based on the nature of the information: structural, behavioral, and mixed. 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. This section presents the survey using content-based classification of ADLs.

#### 2.1.1 Structural ADLs

There are two important aspects to consider to design an ADL: 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

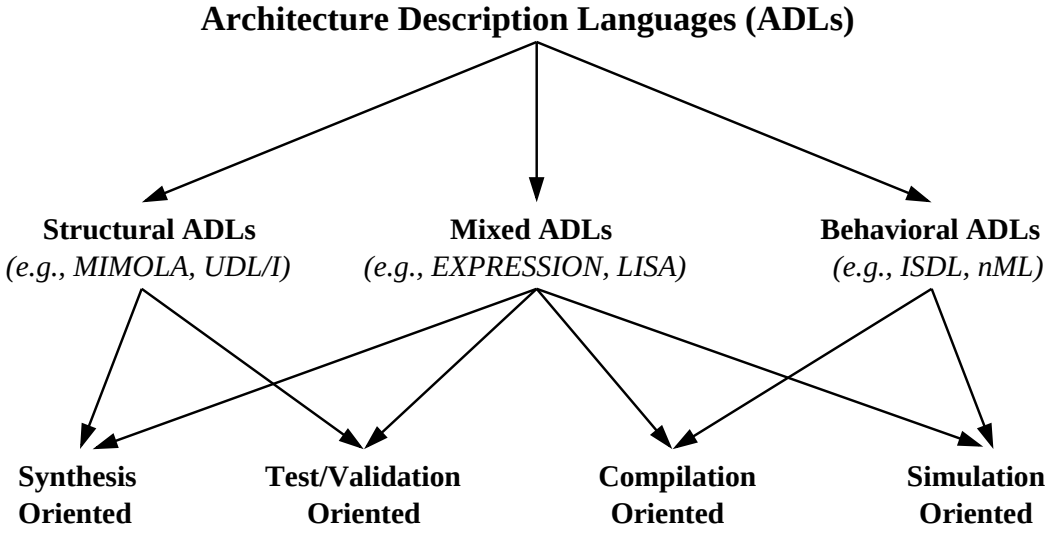


**Figure 3. Taxonomy of ADLs**

gate-level implementation details. Early ADLs are based on RT-level descriptions. This section briefly describes a structural ADL: MIMOLA [33].

**MIMOLA**

MIMOLA [33] 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 [33]. MIMOLA has also been used by the RECORD [33] 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 [33]:

```
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 [33]:

```
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.

### 2.1.2 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 [16] and ISDL [8].

#### 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 [5], and instruction set simulators Sigh/Sim [6] and CHECKERS. Currently, CHESS/CHECKERS environment is used for automatic and efficient software compilation and instruction-set simulation [12]. 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 [15]. 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. The following nML description shows an example of instruction specification [16]:

```
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. 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.

**ISDL**

Instruction Set Description Language (ISDL) was developed at MIT and used by the Aviv compiler [34] and GENSIM simulator generator [7]. 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 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. 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.

### 2.1.3 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 [13]. 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 backend 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 [21]. 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 [2] describes a processor as a netlist of units and storages to automatically generate RTs based on the netlist [24]. 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 and simulator generation framework [11]. 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.

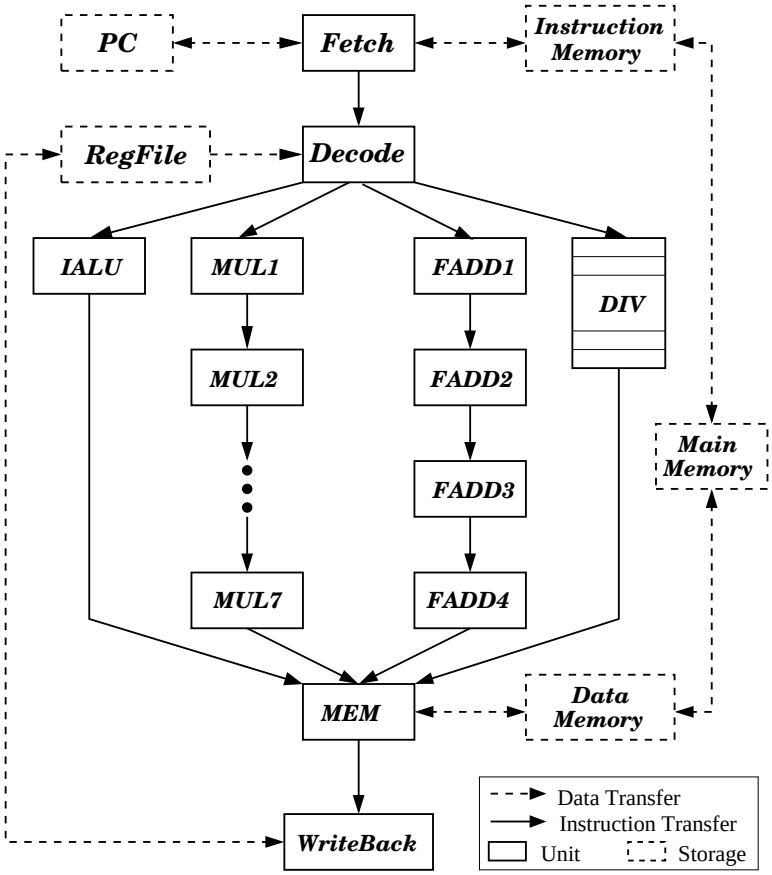


**Figure 4. A VLIW DLX Architecture**

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 [24]. An

example path declaration for the DLX architecture [14] (Figure 4) 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 [26]. 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) [36] was developed at Aachen University of Technology, Germany with a simulator centric view. The language has been used to produce production quality simulators [35]. 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 [17]. 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 4 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 [4] 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. Efficient software toolkit generation is also demonstrated using Mescal Architecture Description Language (MADL) [38]. MADL uses Operation State Machine (OSM) model to describe the operations at the cycle-accurate level. Due to OSM based modeling of operations, MADL provides flexibility in describing a wide range of architectures, simulation efficiency, and ease of extracting model properties for efficient compilation.

### 2.2 Objective-based Classification of ADLs

ADLs have been successfully used as a specification language for processor development. Rapid evaluation of candidate architectures is necessary to explore the vast design and find the best possible design under various design constraints such as area, power, and performance. Figure 5 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. The generated simulator produces profiling data that can be used to evaluate and instruction set, performance of an algorithm, and required size of memory and registers. The generated hardware (synthesizable HDL) model can provide more accurate feedback related to required silicon area, clock frequency, and power consumption of the processor architecture. Contemporary ADLs can be classified into four categories based on the objective: compilation-oriented, simulation-oriented, synthesis-oriented and validation-oriented. In this section we briefly describe the ADLs based on the objective based classification. We primarily discuss the required capabilities in an ADL to perform the intended objective.

#### 2.2.1 Compilation-oriented ADLs

The goal of such an ADL is to enable automatic generation of retargetable compilers. 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. Therefore, behavioral ADLs (e.g., ISDL [8] and nML [16]) are suitable for compiler generation. They capture the instruction-set of the architecture along with some structural information such as program counter and register details. Mixed ADLs are suitable for compiler generation since they capture both structure and behavior of the processor.

There is a balance between the information captured in an ADL and the information necessary for compiler optimizations. Certain ADLs (e.g., AVIV [34] using ISDL, CHESS [5] using nML, and Elcor [21] using MDES) explicitly capture all the necessary details such as instruction-set and resource conflict information. Recognizing that the architecture information needed by the compiler is not always in a form

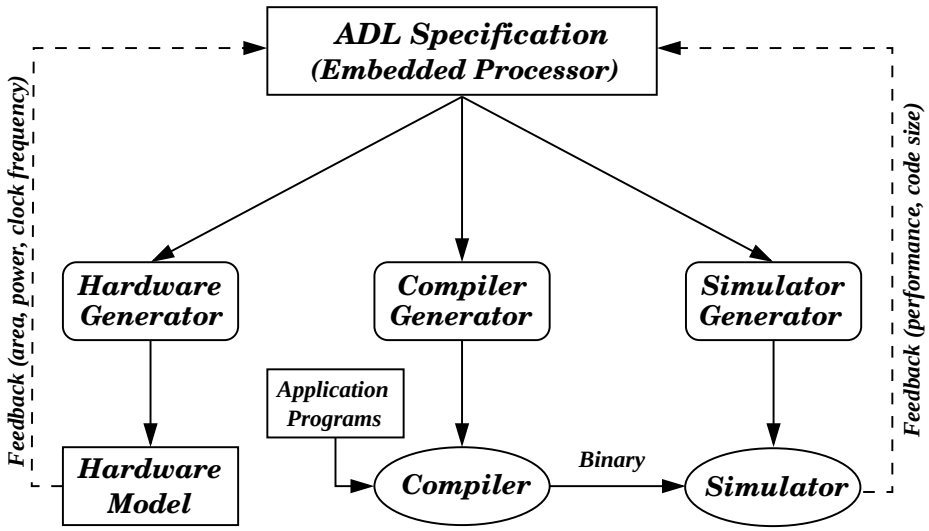


**Figure 5. ADL driven Design Space Exploration**

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 [33], compiler optimizers using MADL [38], retargetable C compiler based on LISA [17], and the EXPRESS compiler using EXPRESSION [2].

### 2.2.2 Simulation-oriented ADLs

Simulation can be performed at various abstraction levels. At the highest level of abstraction, functional simulation (instruction-set simulation) of the processor can be performed by modeling only the instruction-set. Behavioral ADLs can enable generation of functional simulators. The cycle-accurate and phase-accurate simulation models yield more detailed timing information since they are at lower level of abstraction. Structural ADLs are good candidates for cycle-accurate simulator generation.

Retargetability (i.e., ability to simulate a wide variety of target processors) is especially important in the context of customizable processor design. Simulators with limited retargetability are very fast but may not be useful in all aspects of the design process. Such simulators (e.g., HPL-PD [21] using MDES) typically incorporate a fixed architecture template and allow only limited retargetability in the form of parameters such as number of registers and ALUs. Due to OSM based modeling of operations, MADL allows modeling and simulator generation for a wide range of architectures [38]. Based on the simulation model, simulators can be classified into three types: interpretive, compiled, and mixed. Interpretive simulators (e.g., GENSIM/XSIM [7] using ISDL) offers flexibility but slow due to fetch, decode, and execution model for each instruction. Compilation based approaches (e.g., [35] using LISA) reduce the runtime overhead by translating each target instruction into a series of host machine instructions which manipulate the simulated machine state. Recently proposed techniques (JIT-CCS [3] using LISA and IS-CS [20] using EXPRESSION) combines the flexibility of interpretive simulation with the speed of the compiled simulation.

### 2.2.3 Synthesis-oriented ADLs

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 [7] uses ISDL description, and the synthesis tool GO [12] is based on nML. Itoh et al. [18] have

proposed a micro-operation description based synthesizable HDL generation. Mixed languages such as LISA and EXPRESSION capture both structure and behavior of the processor and enables HDL generation [23, 25]. The synthesizable HDL generation approach based on LISA language produces an HDL model of the architecture. The designer has the choice to generate a VHDL, Verilog or SystemC representation of the target architecture [23].

### 2.2.4 Validation-oriented ADLs

ADLs have been successfully used in both academia as well as industry to enable test generation for functional validation of embedded processors. Traditionally, structural ADLs such as MIMOLA [33] are suitable for test generation. Behavioral ADLs such as nML [12] have been used successfully for test generation. Mixed ADLs also enable test generation based on coverage of the ADL specification using EXPRESSION [28, 29, 9] as well as automated test generation for LISA processor models [22]. ADLs have been used in the context of functional verification of embedded processors [32] using a top-down validation methodology as shown in Figure 6. The first step in the methodology is to verify the ADL specification to ensure the correctness of the specified architecture [27]. The validated ADL specification can be used as a golden reference model for various validation tasks including property checking, test generation and equivalence checking. For example, the generated hardware model (reference) can be used to perform both property checking and equivalence checking of the implementation using EXPRESSION ADL [30].

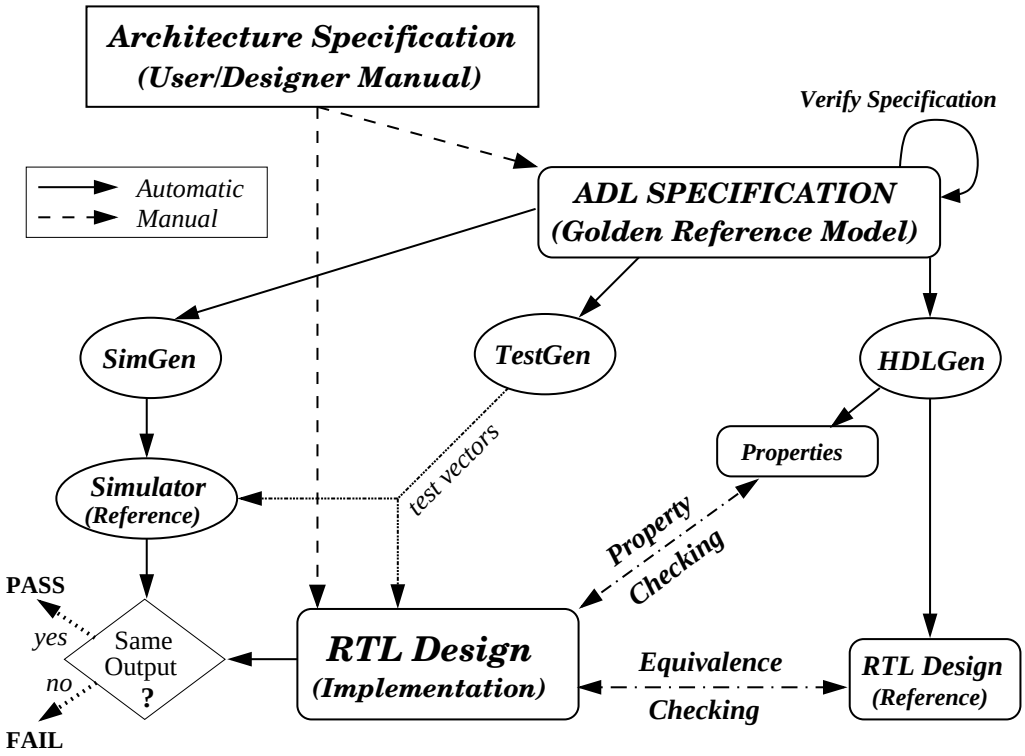


**Figure 6. Top-Down Validation Flow**

## 3 Conclusions

Design of customizable and configurable embedded 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 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. However, a behavioral ADL can be modified to perform the task of a structural ADL (and vice versa). For example, nML is extended by Target Compiler Technologies to perform hardware synthesis and test generation [12]. The later ADLs (LISA, HMDES and EXPRESSION) adopted the mixed approach where the language captures both structure and behavior of the architecture. 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 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.

[3] 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.

[4] 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.

[5] 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 P. Marwedel and G. Goossens, editor, *Code Generation for Embedded Processors*, pages 85–102. Kluwer Academic Publishers, 1995.

[6] 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.

[7] 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.

[8] 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.

[9] H. Koo and P. Mishra. Functional test generation using property decompositions for validation of pipelined processors. In *Proceedings of Design Automation and Test in Europe (DATE)*, 2006.

[10] 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.

[11] http://www.ics.uci.edu/~express. *Exploration framework using EXPRESSION ADL.*

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

[13] 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.

[14] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 2003.

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

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

[17] 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.

[18] M. Itoh and Y. Takeuchi and M. Imai and A. Shiomi. Synthesizable HDL generation for pipelined processors from a micro-operation description. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E83-A(3):394–400, March 2000.

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

[20] 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.

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

[22] O. Luthje. A methodology for automated test generation for LISA processor models. In *Proceedings of Synthesis and System Integration of Mixed Technologies (SASIMI)*, pages 266–273, 2004.

[23] 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.

[24] P. Grun and A. Halambi and N. Dutt and A. Nicolau. RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(4):731–737, August 2003.

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

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

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

[28] 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.

[29] 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)*, pages 678–683, 2005.

[30] 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.

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

[32] Prabhat Mishra and Nikil Dutt. *Functional Verification of Programmable Embedded Architectures: A Top-Down Approach*. Springer, 2005.

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

[34] 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.

[35] 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.

[36] 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.

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

[38] W. Qin, S. Rajagopalan and S. Malik. A formal concurrency model based architecture description language for synthesis of software development tools. In *Proceedings of ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 47–56, 2004.

## Index