Architecture Description Languages for Programmable Embedded Systems

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
University of California, Irvine, CA 92697
dutt@ics.uci.edu

Abstract

Embedded systems present a tremendous opportunity to customize designs by exploiting the application behavior. Shrinking time-to-market, coupled with short product lifetimes create a critical need for rapid exploration and evaluation of candidate architectures. Architecture Description Languages (ADL) enable exploration of programmable architectures for a given set of application programs under various design constraints such as area, power, and performance. The ADL is used to specify programmable embedded systems including processor, coprocessor and memory architectures. The ADL specification is used to generate a variety of software tools and models facilitating exploration and validation of candidate architectures. This chapter surveys the existing ADLs in terms of (a) the inherent features of the languages; and (b) the methodologies they support to enable simulation, compilation, synthesis, test generation, and validation of programmable embedded systems. It concludes with a discussion of relative merits and demerits of the existing ADLs, and expected features of future ADLs.

1 Introduction

Embedded systems are everywhere - 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. Cars are full of them, as are airplanes, satellites, and advanced military and medical equipments. As applications grow increasingly complex, so do the complexities of the embedded computing devices. Figure 1 shows an example embedded system, consisting of programmable components including a processor core, coprocessors, and memory subsystem. The programmable components are used to execute the application programs. Depending on the application domain, the embedded system can have application specific hardwares, interfaces, controllers, and peripherals. The programmable components, consisting of

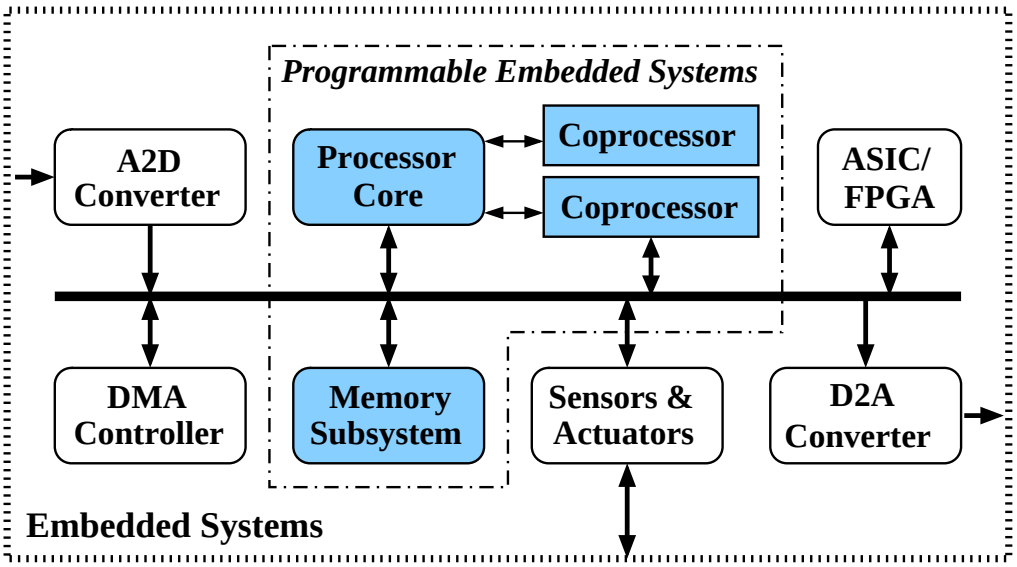

Figure 1. An example embedded system

a processor core, coprocessors, and memory subsystem, are referred as *programmable embedded systems*. They are also referred as *programmable architectures*.

As embedded systems become ubiquitous, there is an urgent need to facilitate rapid design space exploration (DSE) of programmable architectures. This need for rapid DSE becomes even more critical given the dual pressures of shrinking time-to-market and ever-shrinking product lifetimes. Architecture Description Languages (ADL) are used to perform early exploration, synthesis, test generation, and validation of processor-based designs as shown in Figure 2. ADLs are used to specify programmable architectures. The specification can be used for generation of a software toolkit including the compiler, assembler, simulator, and debugger. 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 applications. The ADL specification can also be used for generating hardware prototypes under design constraints such as area, power, and clock speed. Several researches have shown the usefulness of ADL-driven generation of functional test programs and test interfaces. The specification can also be used to generate device drivers for real-time operating systems [77].

Previously, researchers have surveyed architecture description languages for retargetable compilation [82], and systems-on-chip design [24]. Qin et al. [82] surveyed the existing ADLs and compared the ADLs to highlight their relative strengths and weaknesses in the context of retargetable compilation. Tomiyama et al. [24] classified existing ADLs into four categories based on their main objectives: synthesis, compiler generation, simulator generation and validation. This chapter presents a comprehensive survey of existing

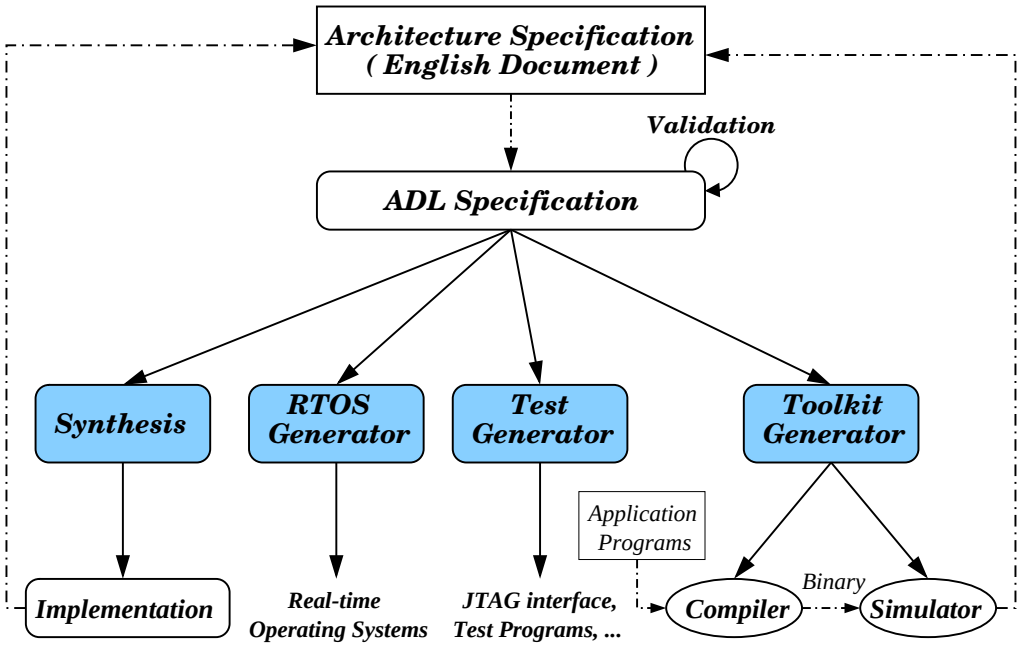

Figure 2. ADL-driven exploration, synthesis, and validation of programmable architectures

ADLs and the accompanying methodologies for programmable embedded systems design.

The rest of the chapter is organized as follows. Section 2 describes how ADLs differ from other modeling languages. Section 3 surveys the existing ADLs. Section 4 presents the ADL-driven methodologies on software toolkit generation, hardware synthesis, exploration, and validation of programmable embedded systems. This study forms the basis for comparing the relative merits and demerits of the existing ADLs in Section 5. Finally, Section 6 concludes this chapter with a discussion on expected features of future ADLs.

2 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 ([45], [68]). 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. This chapter surveys hardware ADLs. The concept of using machine description languages for specification of architectures has been around for a long time. Early ADLs such as ISPS [41] were used for simulation, evaluation, and synthesis of computers and other digital systems. This chapter surveys contemporary 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?

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. Section 5 will re-visit this issue in light of the survey results.

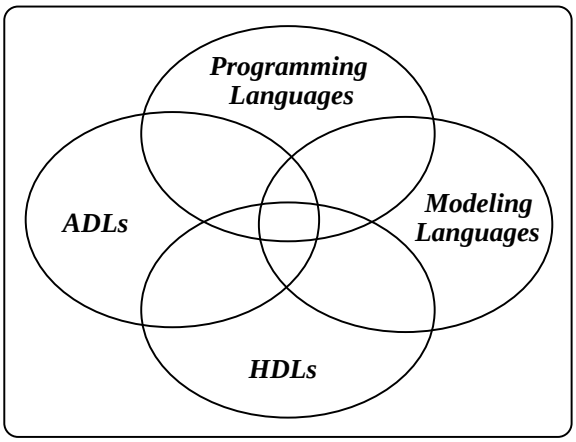

Figure 3. ADLs versus non-ADLs

3 The ADL Survey

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.

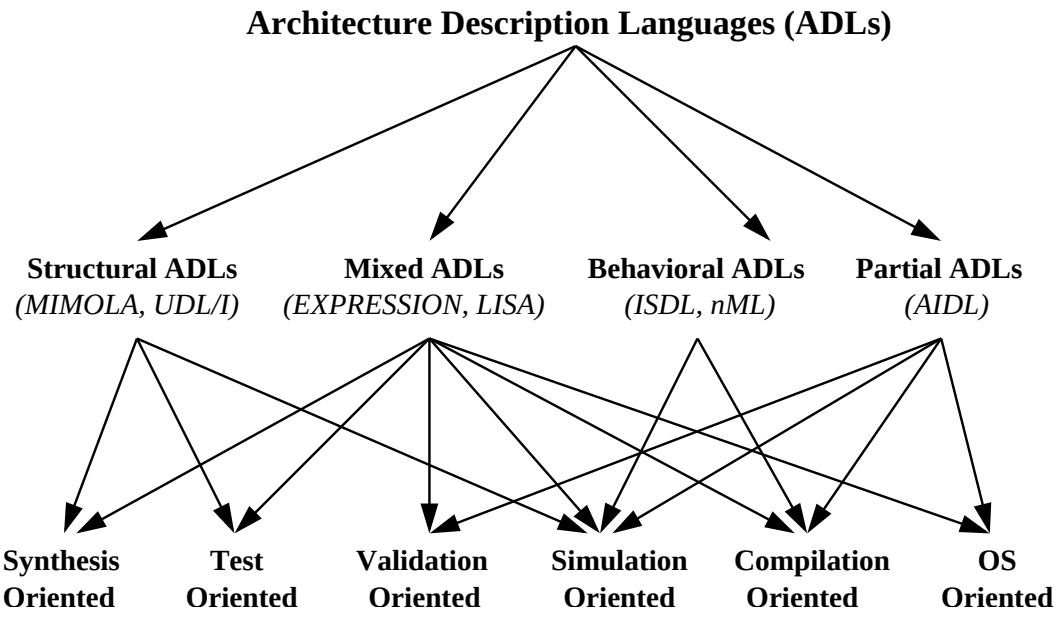

Figure 4. Taxonomy of ADLs

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.

3.1 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 two structural ADLs: MIMOLA [72] and UDL/I [22].

MIMOLA

MIMOLA [72] 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 [72]. MIMOLA has also been used by the RECORD [71] 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 [72]:

```
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 [72]:

```
MODULE ALU
   (IN inp1, inp2: (31:0);
    OUT outp: (31:0);
    IN ctrl: (1:0);
   )
CONBEGIN
       outp <- CASE ctrl OF
           0: inp1 + inp2 ;
           1: inp1 - inp2 ;
           2: inp1 AND inp2 ;
           3: inp1 ;
           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.

UDL/I

Unified design language, UDL/I is [22] developed as a hardware description language for compiler generation in COACH ASIP design environment at Kyushu University, Japan. UDL/I is used for describing processors at an RT-level on a per-cycle basis. The instruction-set is automatically extracted from the UDL/I description [23], and then it is used for generation of a compiler and a simulator. COACH assumes simple RISC processors and does not explicitly support ILP or processor pipelines. The processor description is synthesizable with the UDL/I synthesis system [25]. The major advantage of the COACH system is that it requires a single description for synthesis, simulation, and compilation. Designer needs to provide hints to locate important machine states such as program counter and register files. Due to difficulty in instruction-set extraction (ISE), ISE is not supported for VLIW and superscalar architectures.

Structural ADLs enable flexible and precise micro-architecture descriptions. The same description can be used for hardware synthesis, test generation, simulation and compilation. However, it is difficult to extract instruction-set without restrictions on description style and target scope. Structural ADLs are more suitable for hardware generation than retargetable compilation.

3.2 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 four behavioral ADLs: nML [36], ISDL [21], Valen-C [6] and CSDL ([46], [47]).

nML

nML is an instruction-set oriented ADL proposed at Technical University of Berlin, Germany. nML has been used by code generators CBC [2] and CHESS [15], and instruction set simulators Sigh/Sim [17] and CHECKERS. Currently, CHESS/CHECKERS environment is used for automatic and efficient software compilation and instruction-set simulation [29].

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 [32]. 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 [21] and TDL [14]. The following nML description shows an example of instruction specification [36]:

```
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 [37].

ISDL

Instruction Set Description Language (ISDL) was developed at MIT and used by the Aviv compiler [74] and GENSIM simulator generator [19]. 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.

Valen-C

Valen-C is an embedded software programming language proposed at Kyushu University, Japan [6, 7]. Valen-C is an extended C language which supports explicit and exact bit-width for integer type declarations. A retargetable compiler (called Valen-CC) has been developed that accepts C or Valen-C programs as an input and generates the optimized assembly code. Although Valen-CC assumes simple RISC architectures, it has retargetability to some extent. The most interesting feature of Valen-CC is that the processor can have any datapath bit-width (e.g., 14 bits or 29 bits). The Valen-C system aims at optimization of datapath width. The target processor description for Valen-CC includes the instruction set consisting of behavior and assembly syntax of each instruction as well as the processor datapath width. Valen-CC does not explicitly support processor pipelines or ILP.

CSDL

Computer system description languages (CSDL) is a family of machine description languages developed for the Zephyr compiler infrastructure at University of Virginia. CSDL consists of four languages: SLED [47], λ-RTL, CCL, and PLUNGE. SLED describes assembly and binary representations of instructions [47], while λ-RTL describes the behavior of instructions in a form of register transfers [46]. CCL specifies the convention of function calls [35]. PLUNGE is a graphical notation for specifying the pipeline structure.

Similar to ISDL, SLED (Specification Language for Encoding and Decoding) uses a hierarchical model for machine instruction. SLED models an instruction (binary representation) as a sequence of tokens, which are bit vectors. Tokens may represent whole instructions, as on RISC machines, or parts of instructions, as on CISC machines. Each class of token is declared with multiple fields. The construct patterns help to group the fields together and to bind them to binary values. The directive constructors help to connect

the fields into instruction words. Similar to nML, SLED enumerates legal combinations of fields. There is neither a notion of hardware resources nor explicit constraint descriptions. Therefore, without significant extension, SLED is not suitable for use in VLIW instruction word description [82].

To reduce description effort, λ-RTL was developed. A λ-RTL description will be translated into register-transfer lists for the use of *vpo* (very portable optimizer). According to the developers [46], λ-RTL is a high order, strongly typed, polymorphic, pure functional language based largely on Standard ML [73]. It has many high level language features such as name space (through the module and import directives) and function definition. Users can even introduce new semantics and precedence to basic operators.

In general, the behavioral languages have one feature in common: hierarchical instruction set description based on attribute grammar [32]. This feature simplifies the instruction set description by sharing the common components between operations. However, the capabilities of these models are limited due to the lack of detailed pipeline and timing information. It is not possible to generate cycle accurate simulators without certain assumptions regarding control behavior. Due to lack of structural details, it is also not possible to perform resource-based scheduling using behavioral ADLs.

3.3 Mixed ADLs

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

FlexWare

FlexWare is a CAD system for DSP or ASIP design [67]. The FlexWare system includes the CodeSyn code generator and the Insulin simulator. Both behavior and structure are captured in the target processor description. The machine description for CodeSyn consists of three components: instruction set, available resources (and their classification), and an interconnect graph representing the datapath structure. The instruction set description is a list of generic processor macro instructions to execute each target processor instruction. The simulator uses a VHDL model of a generic parameterizable machine. The parameters include bit-width, number of registers, ALUs, and so on. The application is translated from the user-defined target instruction set to the instruction set of the generic machine. Then, the code is executed on the generic machine. It is not clear how the resource conflicts between instructions are specified.

HMDES

Machine description language HMDES was developed at University of Illinois at Urbana-Champaign

for the IMPACT research compiler [30]. 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 [44]. MDES permits description of memory systems, but limited to the traditional hierarchy, i.e., register files, caches, and main memory.

TDL

Target description language TDL [14] was developed at Saarland University, Germany. The language is used in a retargetable postpass assembly-based code optimization system called PROPAN [13]. A TDL description contains four sections: resource, instruction set, constraints, and assembly format.

TDL offers a set of predefined resource types whose properties can be described by a predefined set of attributes. The predefined resource types comprise functional units, register sets, memories and caches.

Attributes are available to describe the bit-width of registers, their default data type, the size of a memory, its access width, and alignment restrictions. The designer can extend the domain of the predefined attributes and declare user-defined attributes if additional properties have to be taken into account.

Similar to behavioral languages, the instruction-set description of TDL is based on attribute grammar [32]. TDL supports VLIW architectures, so it distinguishes operation and instruction. The instruction-set section also contains definition of operation classes that groups operations for the ease of reference. TDL provides a non-terminal construct to capture common components between operations.

Similar to ISDL, TDL uses Boolean expressions for constraint modeling. A constraint definition includes a premise part followed by a rule part, separated by a colon. The following code segment describes constraints in TDL [14]:

```
op in {C0}: op.dst1 = op.src1;
op1 in {C1} & op2 in {C2}: !(op1 &&op2);
```

The first one enforces the first source operand to be identical to the destination operand for all operations of the operation class C0. The second rule prevents any operation of operation class C1 to be executed in parallel with an operation of operation class C2.

The assembly section deals with syntactic details of the assembly language such as instruction or operation delimiters, assembly directives, and assembly expressions. TDL is assembly-oriented and provides a generic modeling of irregular hardware constraints. TDL provides a well-organized formalism for VLIW DSP assembly code generation.

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 [4] describes a processor as a netlist of units and storages to automatically generate RTs based on the netlist [53]. 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 [3]) and simulator (SIMPRESS [8]) 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 [27].

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.

The pipeline/data-transfer path subsection describes the netlist of the processor. The *pipeline path*

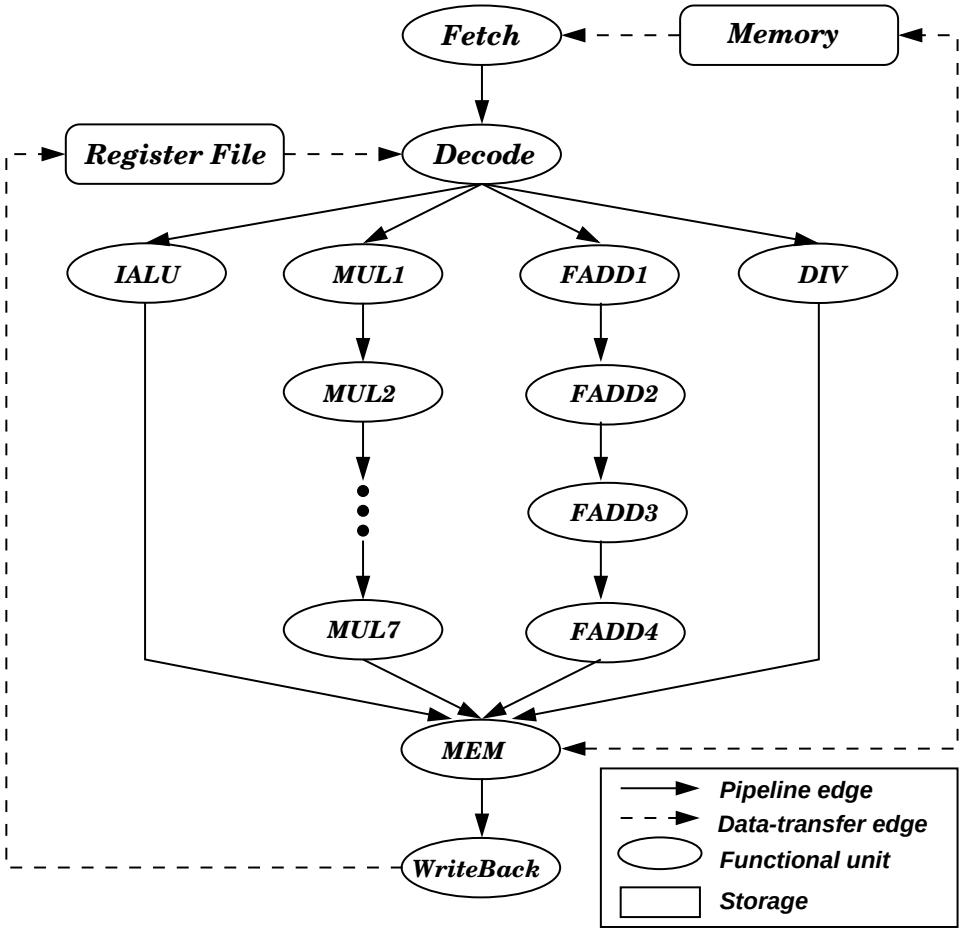

Figure 5. The DLX Architecture

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 [53]. An example path declaration for the DLX architecture [31] (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 [59, 66]. Memory aware compilers can exploit the detailed information to hide the latency of the lengthy memory operations [54].

In general, EXPRESSION captures the data path information in the processor. The control path is not explicitly modeled. Also, the VLIW instruction composition model is simple. 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) [81] was developed at Aachen University of Technology, Germany with a simulator centric view. The language has been used to produce production quality simulators [75]. 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 [38, 51].

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:

- The CODING section describes the binary image of the instruction word.
- The SYNTAX section describes the assembly syntax of instructions.
- The SEMANTICS section specifies the instruction-set semantics.

- The BEHAVIOR section describes components of the behavioral model.
- The ACTIVATION section describes the timing of other operations relative to the current operation.
- The DECLARE section contains local declarations of identifiers.

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 [12] 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.

3.4 Partial ADLs

The ADLs discussed so far captures complete description of the processor's structure, behavior or both. There are many description languages that captures partial information of the architecture needed to perform specific task. This section describes two such ADLs.

AIDL is an ADL developed at University of Tsukuba for design of high-performance superscalar processors [78]. 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.

PEAS-I is a CAD system for ASIP design supporting automatic instruction set optimization, compiler generation, and instruction level simulator generation [33]. In the PEAS-I system, the GNU C compiler is used, and the machine description of GCC is automatically generated. Therefore, there exists no specific ADL in PEAS-I. Inputs to PEAS-I include an application program written in C and input data to the program. Then, the instruction set is automatically selected in such a way that the performance is maximized or the gate count is minimized. Based on the instruction set, GNU CC and an instruction level simulator is automatically retargeted.

4 ADL driven Methodologies

The survey of ADLs is incomplete without a clear understanding of the supported methodologies. This section investigates the contribution of the contemporary ADLs in the following methodologies:

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

4.1 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.

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 [21], Valen-C [6], MIMOLA [71], LISA [81], nML [36], Sim-nML [80], EXPRESSION [4], and industry: ARC [11], Axys [26], RADL [12], Target [29], Tensilica [79], MDES [44].

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.

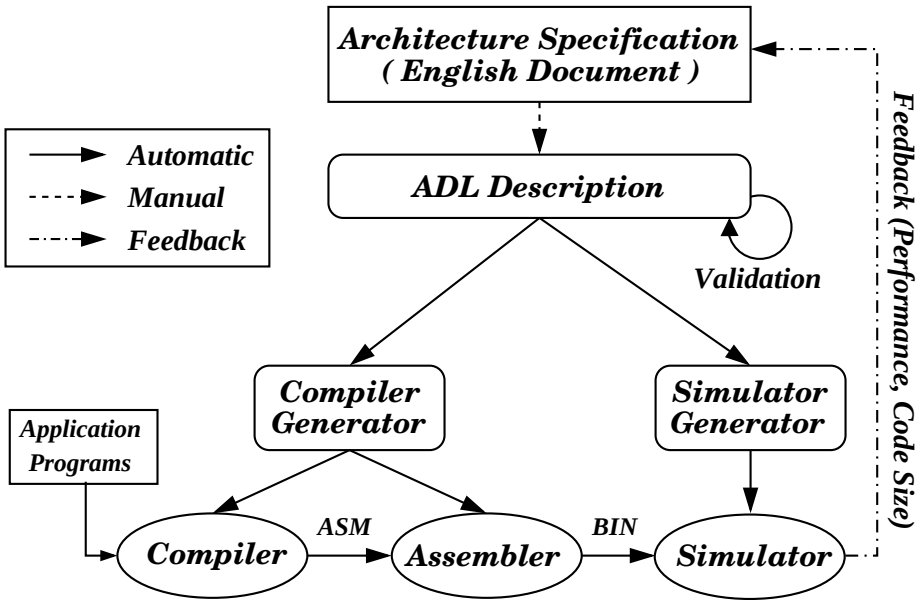

Figure 6. ADL driven Design Space Exploration

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 [53].

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.

4.1.1 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 [55].

The compilation process can be broadly broken into two steps: analysis and synthesis [1]. 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 [76]. 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 [6] and the GNU-based C/C++ compiler from Tensilica Inc. [79]. 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 [74] compiler using ISDL, CHESS [15] using nML, and Elcor [44] 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 [71], retargetable C compiler based on LISA [38], and the EXPRESS compiler using EXPRESSION [4]. 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 [38]. 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.

4.1.2 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 [44] 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 simula-

tors include SIMPRESS [8] using EXPRESSION, and GENSIM/XSIM [19] 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. [34] and Pees et al. [75]. Examples of dynamic compiled code simulators include the Shade simulator [70], and the Embra simulator [16].

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 [9] and IS-CS [43]) 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 [42, 43].

4.2 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 [79], Jazz [28], and PEAS [39]. Xtensa [79] 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 [28] 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 [39] 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 [20] uses ISDL description, and the synthesis tool GO [29] is based on nML. Itoh et al. [40] 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.

Mixed languages such as LISA and EXPRESSION capture both structure and behavior of the processor. The synthesizable HDL generation approach based on LISA language [48] produces an HDL model of the architecture. The designer has the choice to generate a VHDL, Verilog or SystemC representation of the target architecture [49]. The HDL generation methodology presented by Mishra et al. [56] combines the advantages of the processor template based environments and the language based specifications using EXPRESSION ADL.

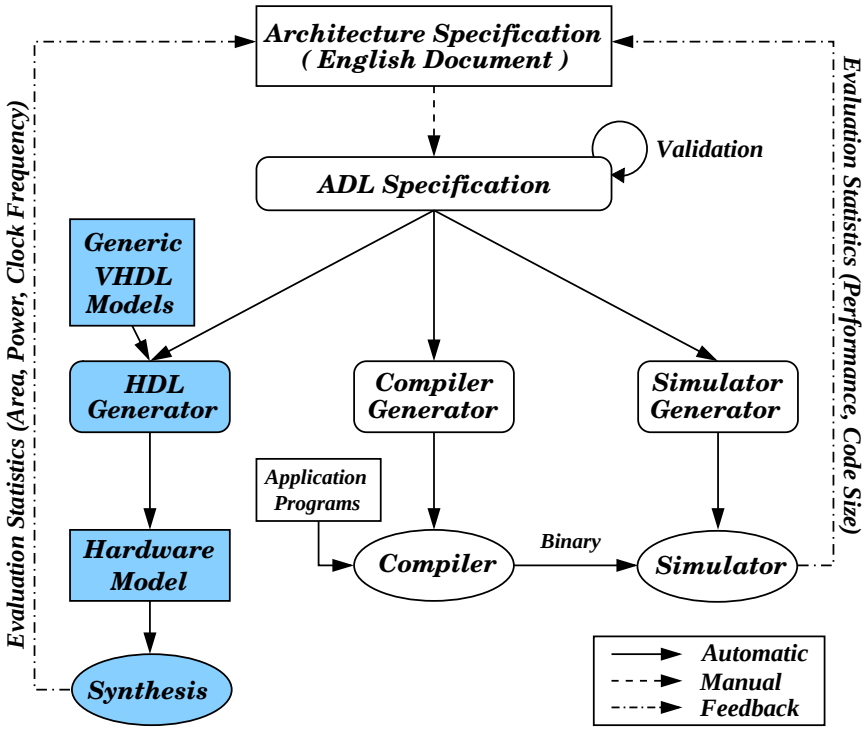

Figure 7. ADL-driven Implementation Generation and Exploration

4.3 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.

Mishra et al. [69] 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 embedded systems 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.

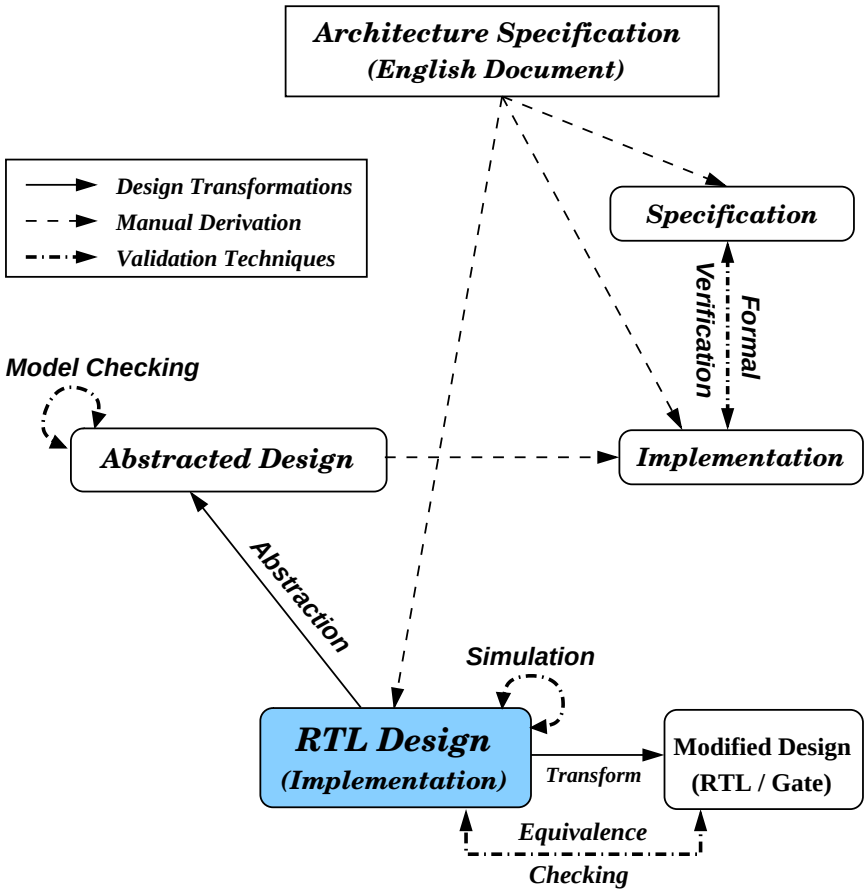

Figure 8. Traditional Bottom-Up Validation Flow

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 [57, 60].

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 [58, 64].

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:

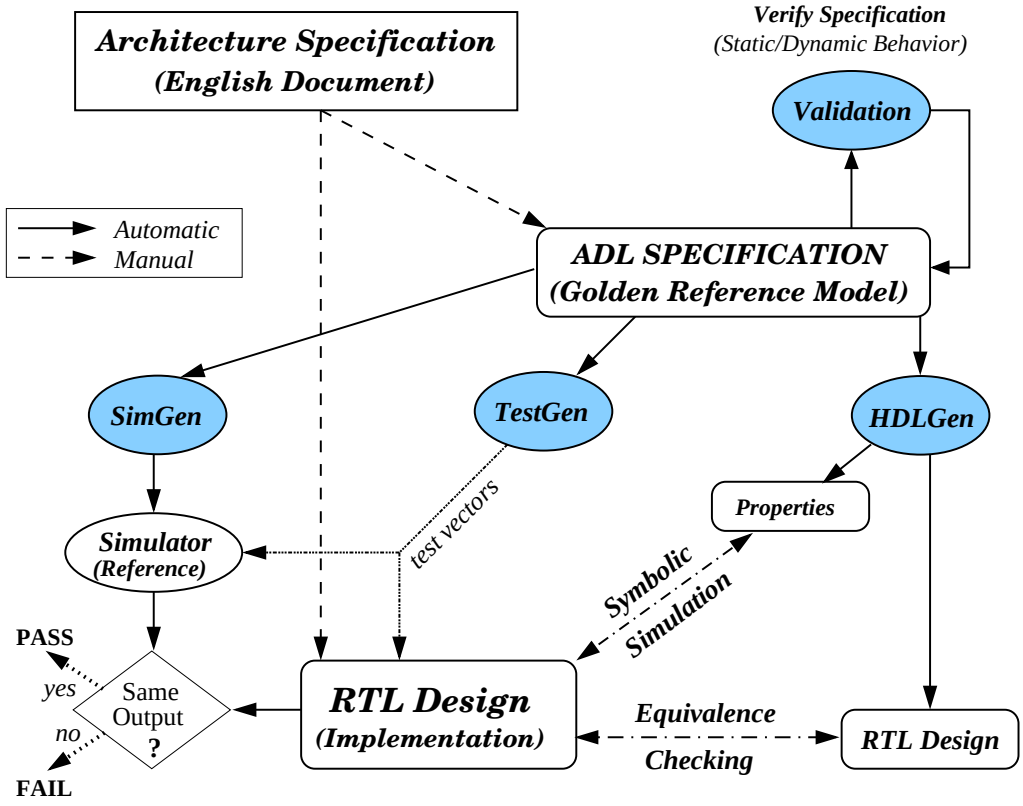

Figure 9. Top-Down Validation Flow

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 [72], EXPRESSION [61], and nML [29]. A model checking based approach is used to automatically generate functional test programs from the processor specification using EXPRESSION ADL [61]. 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 [65]. 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 [65].

Table 1. Comparison between different ADLs

	MIMOLA	UDL/I	nML	ISDL	HMDES	EXPRESSION	LISA	RADL	AIDL
Compiler Generation	√	√	√	√	√	√	√		
Simulator Generation	√	√	√	√	√	√	√	√	√
Cycle-accurate Simulation	√	△		√	√	√	√	√	△
Formal Verification						△			△
Hardware Generation	√	√		△		√	√		△
Test Generation	√		√			√			
JTAG Interface Generation							√		
Instruction-set Information			√	√	√	√	√	√	
Structural Information	√	√			√	√	√	√	
Memory Information					△	√	√		

√ Supported △ Supported with restrictions

5 Comparative Study

Table 1 compares the features of contemporary ADLs in terms of their support for compiler generation, simulator generation, test generation, synthesis, and formal verification. Also, information captured by the ADLs is compared.

Since MIMOLA and UDL/I are originally HDLs, their descriptions are synthesizable and can be simulated using HDL simulators. MIMOLA appears to be successful for retargetable compilation for DSPs with irregular datapaths. However, since its abstraction level is rather low, MIMOLA is laborious to write. COACH (uses UDL/I) supports generation of both compilers and simulators. nML and ISDL support ILP compiler generation. However, due to the lack of structural information, it is not possible to automatically detect resource conflicts between instructions. MDES supports simulator generation only for the HPL-PD processor family. EXPRESSION has ability to automatically generate ILP compilers, reservation tables, and cycle-accurate simulators. Furthermore, description of memory hierarchies is supported. LISA and RADL were originally designed for simulator generation. AIDL descriptions are executable on the AIDL simulator, and does not support compiler generation.

From the above comparison it is obvious that ADLs should capture both behavior (instruction set) and structure (net-list) information in order to generate high-quality software toolkit automatically and efficiently. Behavior information which is necessary for compiler generation should be explicitly specified

for mainly two reasons. First, instruction set extraction from net-lists described in synthesis-oriented ADLs or HDLs does not seem to be applicable to a wide range of processors. Second, synthesis-oriented ADLs or HDLs are generally tedious to write for the purpose of DSE. Also, structure information is necessary not only to generate cycle-accurate simulators but also to generate ILP constraints which are necessary for high-quality ILP compiler generation.

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, time, and power) 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.

6 Conclusions

In the past, an ADL was designed to serve a specific purpose. For example, MIMOLA and UDL have features similar to a hardware description language and were used mainly for synthesis of processor architectures. Similarly, LISA and RADL were designed for simulation of processor architectures. Likewise, MDES and EXPRESSION were designed mainly for generating retargetable compilers.

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.

At present, 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 [29] to perform hardware synthesis and test generation. Similarly, LISA language has been used for hardware generation [5, 49], instruction encoding synthesis [10], and JTAG interface generation [50]. Likewise, EXPRESSION has been used for hardware generation [56], instruction-set synthesis [52], test generation [61, 62], and specification validation [60, 64].

The majority of the ADLs were designed mainly for processor architectures. MDES have features for specifying both processor and memory architectures. EXPRESSION allows specification of processor,

memory and co-processor architectures [63]. Similarly, the language elements of LISA enable the description of processor, memory, peripherals, and external interfaces [18, 50].

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.

7 Acknowledgments

This work was partially supported by NSF grants CCR-0203813 and CCR-0205712. We thank Prof. Alex Nicolau and members of the ACES laboratory for their helpful comments and suggestions. We also thank Mr. Anupam Chattopadhyay for his constructive comments and feedbacks.

References

[1] A. Aho and R. Sethi and J. Ullman. Compilers: Principles, techniques and tools. Addition-Wesley, 1986.

[2] 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.

[3] 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.

[4] 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.

[5] A. Hoffmann and O. Schliebusch and A. Nohl and G. Braun and O. Wahlen and H. Meyr. A methodology for the design of application specific instruction set processors (asip) using the machine description language LISA. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 625–630, 2001.

[6] 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.

[7] A. Inoue and H. Tomiyama and H. Okuma and H. Kanbara and H. Yasuura. Language and compiler for optimizing datapath widths of embedded systems. *IEICE Trans. Fundamentals*, E81-A(12):2595–2604, 1998.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] D. Kastner. *Retargetable Postpass Optimization by Integer Linear Programming*. PhD thesis, Saarland University, Germany, 2000.

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

[15] 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.

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

[17] 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.

[18] G. Braun, A. Nohl, W. Sheng, J. Ceng, M. Hohenauer, H. Scharwaechter, R. Leupers, H. Meyr. A novel approach for flexible and consistent ADL-driven ASIP design. In *Proceedings of Design Automation Conference (DAC)*, pages 717–722, 2004.

[19] 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.

[20] 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.

[21] 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.

[22] H. Akaboshi. *A Study on Design Support for Computer Architecture Design*. PhD thesis, Dept. of Information Systems, Kyushu University, Japan, Jan 1996.

[23] H. Akaboshi and H. Yasuura. Behavior extraction of MPU from HDL description. In *Proceedings of Asia Pacific Conference on Hardware Description Languages (APCHDL)*, 1994.

[24] 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.

[25] http://pjro.metsa.astem.or.jp/udli. *UDL/I Simulation/Synthesis Environment*, 1997.

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

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

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

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

[30] 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.

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

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

[33] J. Sato and A. Alomary and Y. Honma and T. Nakata and A. Shiomi and N. Hikichi and M. Imai. Peas-i: A hardware/software codesign systems for ASIP development. *IEICE Trans. Fundamentals*, E77-A(3):483–491, 1994.

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

[35] M. Bailey and J. Davidson. A formal model and specification language for procedure calling conventions. In *Proceedings of Principle of Programming Languages (POPL)*, pages 298–310, 1995.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

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

[45] 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.

[46] N. Ramsey and J. Davidson. Specifying instructions' semantics using λ-rt. Technical Report , University of Virginia, July 1999.

[47] N. Ramsey and M. Fernandez. Specifying representations of machine instructions. In *Proceedings of ACM TOPLAS*, May 1997.

[48] 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.

[49] O. Schliebusch and A. Hoffmann and A. Nohl and G. Braun and H. Meyr. Architecture implementation using the machine description language LISA. In *Proceedings of Asia South Pacific Design Automation Conference (ASPDAC) / International Conference on VLSI Design*, pages 239–244, 2002.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

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

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

[57] P. Mishra and H. Tomiyama and A. Halambi and P. Grun and N. Dutt and A. Nicolau. Automatic modeling and validation of pipeline specifications driven by an architecture description language. In *Proceedings of Asia South Pacific Design Automation Conference (ASPDAC) / International Conference on VLSI Design*, pages 458–463, 2002.

[58] P. Mishra and H. Tomiyama and N. Dutt and A. Nicolau. Automatic verification of in-order execution in microprocessors with fragmented pipelines and multicycle functional units. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 36–43, 2002.

[59] 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.

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

[61] 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.

[62] 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.

[63] P. Mishra and N. Dutt and A. Nicolau. Functional abstraction driven design space exploration of heterogeneous programmable architectures. In *Proceedings of International Symposium on System Synthesis (ISSS)*, pages 256–261, 2001.

[64] 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.

[65] 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.

[66] P. Mishra and P. Grun and N. Dutt and A. Nicolau. Processor-memory co-exploration driven by an architectural description language. In *Proceedings of International Conference on VLSI Design*, pages 70–75, 2001.

[67] P. Paulin and C. Liem and T. May and S. Sutarwala. FlexWare: A flexible firmware development environment for embedded systems. In *Prof. of Dagstuhl Workshop on Code Generation for Embedded Processors*, pages 67–84, 1994.

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

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

[70] 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.

[71] R. Leupers and P. Marwedel. Retargetable generation of code selectors from HDL processor models. In *Proceedings of European Design and Test Conference (EDTC)*, pages 140–144, 1997.

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

[73] R. Milner and M. Tofte and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1989.

[74] 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.

[75] 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.

[76] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.

[77] 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.

[78] 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.

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

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

[81] 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.

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